Ensemble Methods

Aishwarya Ganesh Murthy

Submitted for the Degree of Master of Science in

Data Science and Analytics

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

September 1, 2015

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14098

Student Name: Aishwarya Ganesh Murthy

Date of Submission: 1 September 2015

Signature:

Acknowledgements

I would like to express my sincere thanks and gratitude to my supervisor Prof. Volodya Vovk, who has guided and refined my work through this project activity. His subject expertise, his intuition on the work areas and his patience as my guide played a major part in making this project what it is today.

Abstract

Machine Learning Algorithms, especially supervised learning algorithms try to reduce the difference between the predicted output and the actual output for accurate prediction. This difference is the error in the prediction, the main aim of all algorithms is to reduce this error. To reduce the error in a single implementation makes the program more complex. So, a simple programs are formed and to increase the efficiency, all the simple programs are combined together. This is known as the ensemble methods. Ensemble methods are machine learning algorithms, which improve the efficiency by combining all the simple algorithms together. Ensemble methods can be split into two different types such as, averaging methods and boosting methods. In this project, two different averaging methods, such as, bagging and random forest and a boosting method, such as boosting have been implemented on real world datasets in the case regression and the base algorithm for implementation of all these algorithms is decision tree learning, bagging and random forest have been implemented using decision trees and boosting have been implemented using decision stumps (decision tree of depth=1). Averaging methods apply a simple algorithm many times on the same data to produce as many predicted output values as possible and an average of all those values is considered as the final predicted value to calculate the error on the test data of the whole dataset, that is, the mean squared error on the test data (test MSE). Bagging is a simple method in which all the attributes are used to form a decision tree, and at each split in the tree a specific attribute is chosen from all the attributes for taking a decision in the tree. Unlike bagging, random forest chooses a set of attributes from all the attributes present and similarly, an attribute is chosen form that set of attributes for taking a decision in the tree. Boosting methods are slightly different from averaging methods, where the values from which the output label is predicted (residuals) is sequentially updated. As the residuals are updated, the best attribute for splitting the dataset is chosen to predict the output. This project implements all the ensemble technique on the real world datasets and compares the efficiency of the algorithms. The main aim of the project is to find out the best ensemble technique in the case of regression among different ensemble techniques available.

Table of Contents

1 Introduction

Machine learning is a field in computer science, where it explores the construction and study of algorithms, from which we can learn and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions. The model is build and is tried on a part of the whole dataset to evaluate result, this part of the dataset is known as 'training data'. A learning program is given training examples of the form $\{(x_1,y_1),\ldots,(x_n,y_n)\}$ for some unknown function $y = f(x)$. The xi values are typically vectors of the form $(x_{i,1}, x_{i,2}, \ldots, x_{i,m})$. The y values are typically drawn from discrete set of classes $\{1,\ldots,K\}$ in the case of classification (qualitative or categorical data) or regression (quantitative or numerical data). From the results, the best model is chosen and is applied on test data to check for the error in the prediction model.

The best model for the given set of data is chosen by comparing the predicted value with the actual value of the output y_i. This method of knowing the actual output and to predict for a possibility of output to be near the actual output is known as **supervised learning**. The main aim of supervised learning is to reduce the gap between the predicted output and the actual output. The absolute difference between the actual and the predicted value of the label in the data set is the error formed in the data. This error value is calculated in the 'test data', which is the remaining data in the dataset other than the training data. The error calculated on the test data, known as the test MSE in case of regression and test error rate in case of classification, should be minimum.

Groups of people can often make better decisions than individuals, especially when group members each come in with their own biases. Hence, we combine and produce a model for prediction leads to **'Ensemble Methods'**. Ensemble techniques are a very powerful tool in Machine Learning. They have proven themselves to be useful for improving the performance of machine learning techniques, and can be applied to an existing process without changing the understanding of the problem too much. So, instead of applying complex algorithms to reduce the error between the actual and the predicted output, we can combine simple algorithms to produce an effective output in the end.

1.1 Purpose of the Project

The plan for this project was to implement various ensemble techniques such as bagging, random forest and boosting together with some underlying algorithms. In the design and coding exercise, it is an exploration of the various ensemble techniques available. The performance using some of the readily available real world datasets and to compare the results to some others already published is tested. The number of machine learning techniques, ensemble techniques and test datasets is very large, so during implementation it was only realistically possible to attack a subset of the possibilities in this project. Having to choose an underlying algorithm for predicting the values, decision tree learning was chosen as the base algorithm for implementation.

1.2 Background

Decision trees are a popular method for various machine learning tasks, which uses a decision tree as a predictive model, which maps observations about an item to conclude about the item's target value. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees tries to meet the requirements for serving as an off-the-shelf procedure for data mining, because it is invariant under scaling and various other transformations of feature values, is robust to inclusion of irrelevant features, and produces models. Tree models where the target variable can take a finite set of values are called **classification trees**. Decision trees where the target variable can take continuous values or real numbers are called **regression trees**.

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. In data mining, a decision tree describes data but not decisions; rather the resulting classification tree can be an input for decision making. [1]

A decision tree is a simple representation for classifying examples, that is, stratifying or segmenting the object space or the set of attributes ($X_1, X_2 \ldots X_P$) into a set of simple regions. In order to make prediction over a particular region in which it belongs for a particular object, the mean or the mode of the training labels are taken in case of regression trees and the maximum vote gained among the training labels are taken in case of classification trees. In decision tree, each interior node corresponds to one of the input variables, edges of the tree lead to the children for each possible value of the input variable. Each leaf represents a value of the target variable given the values of the input variables represented by the path from the root to the leaf. Thus we try to predict the value of the label from the input attributes. The steps involved in making a decision tree are the following:

1. Split the dataset into two sets, one as training set and other as test set
2. Select the best attribute to split the training data every time a split is considered in the tree, to form the best decision tree as the training model
3. At the leaves of the decision tree, obtain the values of the training label
4. Calculate the test MSE in case of regression and test error rate in case of classification.

The test MSE or the test error rate is reduced much using the decision tree algorithm but it can also lead to a case of under-fitting or over-fitting.

1.2.1 Machine Learning Problems

The output produced is a function of the input samples. If the predicted function ($\hat{y} = \hat{f}(x)$) between the input and the output variable is similar to the original function ($y = f(x)$), then the error also reduces. The final aim is to reduce the error, but there are two different scenarios where the expected error calculated is very high. They are the following:

- **Under-fitting**

  1. The models have high bias, that is, the models are not accurate. They have been approximated to form a simple functional relationship between the input and the output.

  2. The models have small variance, that is, there is a smaller influence of examples in the training set.

- **Over-fitting**

  1. The models have low bias, that is, the models are accurate. The models are flexible enough to fit well to training data.

  2. The models have high variance, that is, the models depends very much on the training data set. When the training sample is changed, the function between the input and the output changes leads to a large variation.

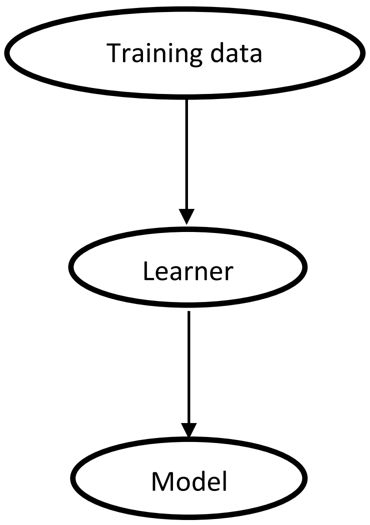


Fig 1: Normal method to find the final model (without the application of ensemble methods)

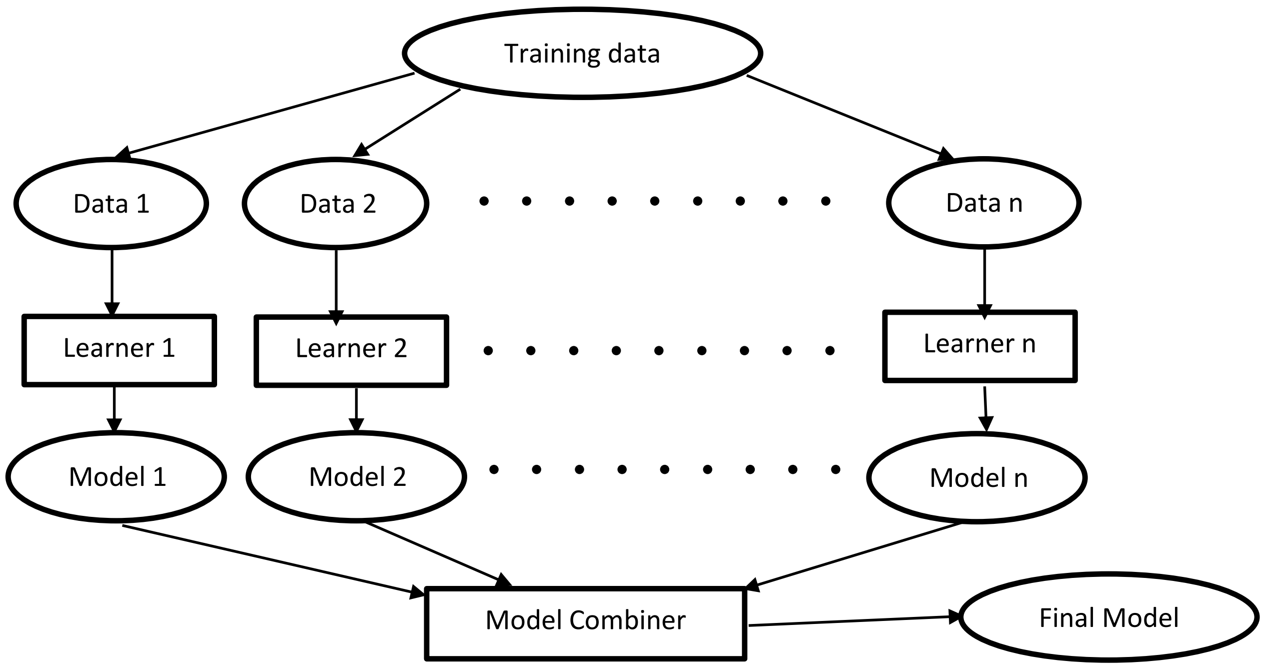


Fig 2: Final model derived using ensemble methods

An ensemble is a set of classifiers whose individual decisions are combined to classify new examples. Ensemble methods help in achieving group work, they are learning models that achieve performance by combining the opinions of multiple learners. The goal of ensemble methods is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability or robustness over a single estimator.

1.3 Need for Ensemble Methods

The learning model becomes too complex to implement when we try to enhance the predictive performance from a single learning model. When we implement a simple model, the accuracy of the prediction might be low and also the hypothesis space will be limited.

Ensemble methods aim at improving the predictive performance of a given statistical learning or model fitting technique. The general principle of ensemble methods is to construct a linear combination of some model fitting method, instead of using a single fit of the method. Thus, ensemble methods is used to combine all the simple models together and average lots of different simple models, the predictive performance will also increase by taking different cases, and also we can fit the simple models well and have large hypothesis space. By implementing ensemble methods, we can also reduce the variance estimator.

The bagging procedure turns out to be a variance reduction scheme, at least for some base procedures. On the other hand, boosting methods are primarily reducing the (model) bias of the base procedure. This already indicates that bagging and boosting are very different ensemble methods.

2 Background Research

2.1 Types of Ensemble Methods

Ensembles methods uses a collection of all models to get better predictive performance than any single model and aggregation of predictions of multiple classifiers improves the accuracy. There exists two different families of ensemble methods, they are the following:

- **Averaging methods**
  Driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced. Examples of averaging methods are bagging, random forest, etc.

- **Boosting methods**
  The base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. Examples of boosting methods are boosting, ada-boost, etc. [2]

2.2 Averaging Methods

Averaging method is an ensemble method in which a fresh sample of the training data set is taken independently. Different learning models are applied on each sample of the training data set. Assume we measure a random variable x with N (μ, σ^2) distribution. If only one measurement x_i is done, then the expected mean of the measurement is μ and the variance is Var (x_i) = σ^2. If random variable x is measured K times (x_1, x_2 … x_k) and the value of μ is estimated as: (x_1+x_2+…+x_k) / K. So, the value of variance reduces as the value becomes the following:

$$\text{Variance of the base model} = [\text{Var}(x_1) + \ldots \text{Var}(x_k)] / K^2 = K\sigma^2 / K^2 = \sigma^2/K$$

2.2.1 Bagging

Bagging is derived from the word '**B**oostrap **Agg**regat**ing**'. It is an example of averaging methods and is the simplest method among the other ensemble methods. The method uses multiple versions of a training set by using the bootstrap, i.e. sampling with replacement. Each of these data sets is used to train a different model such as neural networks, decision trees, etc. Given a training set of size n, create m samples of size n by drawing n examples from the original data, with replacement. The idea behind bootstrap is that instead of generating independent training sets of size n, generate independent samples of size n (with replacement) from the given training set of size n. Bootstrap is often used for estimating the variability of various statistics.

After finding the base model using the training set, in test example, start all trained base models and predict by combining the results of m samples in the training models. In bagging, to get a single value of the output, combining the results is done by taking the simple majority vote in the case of classification and by taking the mean of all numerical values in the case of regression. Each bootstrap sample will on average contain 63.2% of the unique training examples, the rest are replicates.

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. As they provide a way to reduce overfitting, bagging methods work best with strong and complex models. Bagging decreases error by decreasing the variance in the results due to unstable learners, algorithms (like decision trees) whose output can change dramatically when the training data is slightly changed

2.2.1.1 Analysis of Bagging

Bagging is the way decrease the variance of the prediction by generating additional data for training from your original dataset using combinations of samples with repetitions to produce multisets of the same size as your original data. By increasing the size of the training set, model predictive force

cannot be improved but just decrease the variance, narrowly tuning the prediction to expected outcome.
Expected error is formed by the bias and variance of a data item 'x'. The intended output y=f(x) has the bias as the squared discrepancy between averaged estimated and true function. When the input data points of X, say x_1, x_2 .... x_n, etc, are applied, to simplify the obtained output we assign a function in terms of the input data 'X' (say f(X)). The difference between the output data points and the output brought due to the function f(X) is known as bias. The bias must be low to reduce the error between the actual and expected output. It is given by the following formula:

$$\textbf{Bias} = (E[\hat{f}(X)] - E[f(X)])^2$$

The variance is expected divergence of the estimated function vs. its average value. When the input data points of X, say x_1, x_2 .... x_n, etc, are applied, the variation produced between the actual output and the expected output. The variance must be low when the same function is applied to different set of samples of training data. It is given by the following formula:

$$\textbf{Variance} = E[(\hat{f}(X) - E[\hat{f}(X)])^2]$$

The expected error is the sum of bias and variance, the reducible and the irreducible error is formed by the bias and varince of f(X). The values of bias and variance changes the values of expected error. The challenge is to reduce the bias as well as variance together to reduce the expected error in the model. This is achieved in bagging as it decreases variance of the base model without changing the bias.

One of the major problems of all machine learning algorithms is to "know when to stop," i.e., how to prevent the learning algorithm to fit esoteric aspects of the training data that are not likely to improve the predictive validity of the respective model. This case has high variance and low bias. So, by keeping a low bias and high variance simple model, we can apply bagging and obtain a model with low bias and low variance by just averaging the variance obtained in each iteration. This is the problem of over-fitting. Hence bagging typically helps over-fitted base model with low bias and high variance, that is, a model which depends much on the training data as it can help to reduce the variance but it cannot help much if the base model is an under-fitted model, that is, when it has high bias and is robust to changes in training data.

2.2.1.2 Algorithm for Bagging

The steps involved in bagging are the following:

1. Generate B different bootstrapped training sets (say B=1000)
2. Train our prediction algorithm on bth bootstrapped training set in order to get the predicted output $f^{*b}(x)$
3. Average all the predictions to obtain $\hat{f}_{bag}$

$$\hat{f}_{bag}(x) = (1/B) \sum \hat{f}^{*b}(x)$$

4. We can now calculate the test MSE in the case of regression and the test error rate in the case of classification.

2.2.1.2.1 Regression Example

Bagging is particularly useful for decision trees. To apply bagging to regression trees, we construct B regression trees using B bootstrapped training sets, and average the resulting predictions. These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Thus, averaging these B trees reduces the variance.

Let's consider a data set where we have 8 numerical data in the training data. Bagging re-samples the training set 4 times. We get different training samples with replacement, and the output ($\hat{y}$) for each sample is predicted as the mean of the samples.

The mean value of predicted output ($\hat{y}$) is found f_{bag} and the model is applied on the test data set. Then, the value of mean squared error (MSE) over the test data, that is, the test MSE is calculated. The test MSE value will be decreased after bagging proving the fact that the error reduces after bagging. Let's consider the following example below:

A sample of a single classifier on an imaginary set of data	
	(Original) Training-set
Training-Set	1, 2, 3, 4, 5, 6,7,8

The training set consists of 8 numerical values, these values are re-sampled B times (say 4 times) as shown below. The output ($\hat{y}$) is predicted for each sample by taking the mean of those values in the training set. [3]

A sample of Bagging on the same data		
	(Re-Sampled) Training-set	**Predicted Output ($\hat{y}$)**
Training-Set 1	2, 7, 8, 3, 7, 6, 3, 1	4.625
Training-Set 2	7, 8, 5, 6, 4, 2, 7, 1	5
Training-Set 3	3, 6, 2, 7, 5, 6, 2, 2	4.125
Training-Set 4	4, 5, 1, 4, 6, 4, 3, 8	4.375

Instead of calculating the test MSE or the error calculation while trying to predict the model on each training set taken, average of the different predictions are taken together to improve the efficiency of the predicted model. This average ($\hat{f}_{bag}$) is calculated as follows:

$$\hat{f}_{bag} = (1/B) * \sum (\hat{y})$$

where 'B' is the number of times the training set was re-sampled and $\hat{y}$ is the predicted output. So, the value of $\hat{f}_{bag}$ here is obtained by substituting the values of n=4 and the values of $\hat{y}$ obtained above:

$$\hat{f}_{bag} = (1/4)*(4.625 + 5 + 4.125 + 4.375) = 4.531$$

Hence we have got the $\hat{f}_{bag}$, we can find the test MSE by substituting the $\hat{f}_{bag}$ in the place of $\hat{y}$ in the formula for test MSE. Thus, the test MSE is calculated as follows:

$$\textbf{Test MSE} = (1/n) * \sum (y_i - \hat{f}_{bag})^2$$

where 'n' is the size of the test data set

'y_i' is the actual output

'$\hat{f}_{bag}$' is the average of the predicted outputs

2.2.1.2.2 Classification Example

Bagging is used in several approaches in classification but the simplest and the most common approach is for a given test object, record the class predicted by each of the B trees. The overall prediction is the most commonly occurring class among the B predictions, that is, the class that has gained the maximum number of votes among all the other classes.

Let's consider a dataset where we have to predict whether it is going to rain or not according to other prevailing conditions of the weather. If the prediction is 'Going to Rain', the binary value is '1' and if the prediction is 'Not Going to Rain', the binary value is '0'. We get different training samples with replacement, and the output ($\hat{y}$) for each sample is predicted as the maximum vote gained in that sample.

The maximum vote gained or the predicted output is obtained for each training sample. Then the maximum occurrence of a value in the predicted outputs of all the samples is found f_{avg} and the model is applied on the test data set. Then, the value of error rate over the test data, that is, the test error rate is calculated. The test error rate value will be decreased after bagging proving the fact that the error reduces after bagging. Let's consider the following example below:

A sample of a single classifier on an imaginary set of data	
	(Original) Training-set
Training-Set	0, 1

The training set consists of a mixture of two types of binary values (say 0 and 1), these values are re-sampled B times (say 4 times) as shown below. The output ($\hat{y}$) is predicted for each sample by taking the maximum vote of those values in the training set.

A sample of Bagging on the same data		
	(Re-Sampled) Training-set	**Predicted Output ($\hat{y}$)**
Training-Set 1	0, 1, 1, 1, 0, 0, 1, 1	1
Training-Set 2	0, 0, 1, 1, 0, 0, 1, 0	0
Training-Set 3	1, 0, 0, 1, 1, 1, 1, 1	1

Training-Set 4	1, 1, 1, 0, 0, 1, 0, 1	1

Instead of calculating the test MSE or the error calculation while trying to predict the model on each training set taken, maximum vote of the different predictions are taken together to improve the efficiency of the predicted model. This average ($\hat{f}_{bag}$) is calculated as follows:

$$\hat{f}_{bag} = \text{(majority vote in } \hat{y} \text{ found 'B' times)}$$

where 'n' is the number of times the training set was re-sampled and $\hat{y}$ is the predicted output. So, the value of $\hat{f}_{bag}$ here is the maximum vote gained while predicting $\hat{y}$ B times is '1'.

$$\hat{f}_{bag} = \text{majority vote } (1, 0, 1, 1) = 1$$

Hence we have got the $\hat{f}_{bag}$, we can find the test MSE by substituting the $\hat{f}_{bag}$ in the place of $\hat{y}$ in the formula for test MSE. Thus, the test MSE is calculated as follows:

$$\text{Test MSE} = (1/k) * \sum (y_i - \hat{f}_{bag})^2$$

where 'k' is the size of the test data set

'y_i' is the actual output

'f_{avg}' is the average of the predicted outputs

2.2.2 Random Forest

Another type of averaging method is the Random Forests, which is like bagging. Random Forest is the same as bagging where we find the natural balance between the two extremes, that is, high variance and high bias. The idea of Random Forest is to tweak bagged trees by making the trees less dependent.

Similar to bagging, random forest also uses multiple versions of a training set by using the bootstrap, i.e. sampling with replacement. Each of these data sets is used to train a different model such as neural networks, decision trees, etc. Given a training set of size n, create k samples of size n by drawing n examples from the original data, with replacement. The idea behind bootstrap is that instead of generating independent training sets of size n, generate independent samples of size n (with replacement) from the given training set of size n. Bootstrap is often used for estimating the variability of various statistics.

After finding the base model using the training set, in test example, start all trained base models and predict by combining the results of m samples in the training models and, to get a single value of the output, the results are combined by taking the simple majority vote in the case of classification and by taking the mean of all numerical values in the case of regression.

In bagging, we use all the attributes of the dataset to form a decision tree and predict the label but in random forest, we randomly select a few attributes at each split to form a decision tree and predict the label. Consider that the dataset has ‘p’ different attributes, when building decision tree, each time a split in a tree is considered, choose a random sample of ‘m’ attributes as split candidates from the full set of ‘p’ attributes, if m = p, random forest is just bagging. A fresh sample of ‘m’ attributes is taken at each split. Typically, if $m = \sqrt{p}$ number of attributes is used as a choice for attributes at each split, random forest can be implemented properly to form a decision tree and predict the label.

Since random forest have very few parameters to tune and can be used quite efficiently with default parameter settings, that is, they are effectively non-parametric. Random Forests are good to use as a first cut when you don't know the underlying model, or when you need to produce a decent model under severe time pressure. This ease of use also makes Random Forests an ideal tool for people without a background in statistics, allowing lay people to produce fairly strong predictions free from many common mistakes, with only a small amount of research and programming.

Random forest is basically a model which includes averaging algorithms based on randomized decision trees. [In random forest, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model.] [2]

2.2.2.1 Algorithm of Random Forest

The steps involved in random forest are the following:

1. Generate B different bootstrapped training sets (say B=1000)
2. Train our prediction algorithm on bth bootstrapped training set
3. For each split in the decision tree, choose a random sample of m attributes as split candidates from the full set of p attributes.
4. A fresh sample of m attributes is taken at each split in order to get the predicted output $\hat{f}^{*b}(x)$
5. Average all the predictions to obtain $\hat{f}_{bag}$

$$\hat{f}_{bag} = (1/B) * \sum \hat{f}^{*b}(x)$$

6. We can now calculate the test MSE in the case of regression and the test error rate in the case of classification.

2.3 Boosting Methods

Boosting methods are the contrast of averaging methods, here we do not average different trees but we try to predict the label by sequentially updating the values of the estimator and finally reduce the bias of the combined estimator. This method is basically by combining several weak models and by sequential updating the estimator values, we obtain a powerful ensemble.

2.3.1 Boosting for Regression Trees

The algorithm for Boosting Trees evolved from the application of boosting methods to regression trees. The general idea is to compute a sequence of simple trees, where each successive tree is built for the prediction residuals of the preceding tree. At each step of the boosting, a best partitioning of the data is determined, and the deviations of the observed values from the respective means or the residuals for each partition are computed. The next tree will be fitted to the residuals obtained before, to find a partition that will further reduce the residual variance or the error for the data, given the preceding sequence of trees.

A similar approach is for each consecutive simple tree to be built for only a randomly selected subsample of the full data set. In other words, each consecutive tree is built for the prediction residuals (from all preceding trees) of an independently drawn random sample. The introduction of a certain degree of randomness into the analysis in this manner can serve as a powerful safeguard against overfitting (since each consecutive tree is built for a different sample of observations), and yield models (additive weighted expansions of simple trees) that generalize well to new observations, that is, exhibit good predictive validity. This technique is performing consecutive boosting computations on independently drawn samples of observations. [4]

2.3.1.1 Analysis of Boosting

Over-fitting tries to fit a single large decision tree to the data, thus fitting the data hard into the model. This leads to low bias and high variance and averaging methods are a way to reduce the variance but boosting tries to prevent the situation of over-fitting because it fits the model slowly by updating the residuals at each iteration.

Given the current prediction rule, we fit a decision tree to its residuals. We then add a bit of this new decision tree into the prediction rule in order to improve the residuals. Each of the new trees can be rather small, with their size determined by the parameter d, that is, the number of splits in the decision tree. Thus, the prediction $\hat{f}$ in the over-fitted model is improved at the points where it does not perform well.

2.3.1.2 Algorithm for Boosting

Let us consider the input as x_i and the actual output is y_i, where the function $\hat{y} = \hat{f}(x_i)$ and where i = 1….n. The steps involved in boosting are the following:

1. Initialise the values as follows:
   - The prediction $\hat{f}(x_i) = 0$
   - The residuals r_i is initialis to the actual output y_i

2. For b = 1…..B, where b is the number of trees
   - Let there be d splits in the tree, so there will be (d+1) terminal nodes, if d=1, it is known as the decision stump. Fit the decision stump f^b to the

training data (x_i, r_i), i = 1….n. f^b is the decision stump with the smallest training MSE

- update f̂ by adding in a shrunken version of the new stump: $\hat{f}(x) := \hat{f}(x) + \eta f^b(x)$
- update the residuals: $r_i := r_i - \eta f^b(x_i)$, i = 1,...,n

3. Update the prediction rule $\hat{f}(x)$ as

$$\hat{f}(x) = \sum \eta \hat{f}^b(x)$$

4. The test MSE value is computed for the above prediction rule using the following formula:

$$\text{Test MSE} = (1/m) \sum (y_j - \sum \eta \hat{f}^b(x_j))^2$$

where, 'm' is the size of the test set

2.3.1.2.1 Example

Let's consider a data set where we have 5 numerical data in the training data. Boosting re-samples the training set B times (say B=4). We get different training dataset by updating the residuals, and the output (ŷ) for each set is predicted as the mean of the set. The residuals are first initialised to the actual output value (y_i).

The value of mean squared error (MSE) over the test data, that is, the test MSE is calculated. The test MSE value will be decreased after boosting proving the fact that the error reduces after boosting. Let's consider the following example below:

A sample of a single classifier on an imaginary set of data	
	(Original) Training-set
Training-Set	1, 2, 3, 4, 5, 6,7,8

The training set consists of 8 numerical values, these values are computed B times (say 4 times) as shown below. The output (ŷ) is predicted for each sample by taking the mean of the residuals that are updated training set. Then the residuals are calculated by subtracting the previous value of residuals to the shrunken version of the predicted output (f̂), that is, to ηf^b, where the value of η is 0.01.

A sample of Bagging on the same data			
	Residuals	**Predicted Output ($\hat{f}^b$)**	**Updated Residuals**
Training-Set 1	2, 7, 8, 3, 7	4.625	1.95, 6.95, 7.95, 2.95, 6.95
Training-Set 2	1.95, 6.95, 7.95, 2.95, 6.95	5.35	1.90, 6.90, 7.90, 2.90, 6.90

Training-Set 3	1.90, 6.90, 7.90, 2.90, 6.90	5.3	1.85, 6.85, 7.85, 2.85, 6.85
Training-Set 4	1.85, 6.85, 7.85, 2.85, 6.85	5.25	1.80, 6.80, 7.80, 2.80, 6.80

The prediction rule is updated by summing up all the shrunken values of the predicted output at each iteration, that is, sum of all ηf^b obtained over B iterations. This prediction is f(x) and it is used to calculate the test MSE.

The following formula is used to calculate the test MSE,

$$\textbf{Test MSE} = (1/m) \sum (y_j - \sum \eta \hat{f}^b(x_j))^2$$

where, 'm' is the size of the test set

So, here the value of m=5 and so f(x) is $\sum \eta f^b$

⇨ ((0.01 x 4.625) + (0.01 x 5.35) + (0.01 x 5.3) + (0.01 x 5.25)) = 0.20525

Subtract the value obtained 0.20525, from the actual output y_i values in the test data set and square it to find the mean value. Thus the mean squared error in the test dataset is found using boosting.

3 Experiment

3.1 Design

The design of the project is to implement all the ensemble methods, that is, the averaging methods like bagging and random forest and also the boosting methods. The averaging methods have a similar approach, so they are implemented in the same manner but the boosting method is slightly different, as it has a different style of working on the dataset. All machine learning algorithms, especially supervised learning algorithms are implemented on the training data and is tested for correctness on the test data. Ensemble methods also follow the method of splitting the entire dataset into training set and test set, and I have implemented the decision tree algorithm on the training data to predict the output label with respect to all the other attributes.

Bagging and Random Forest have been implemented using decision tree algorithm and Boosting is implemented using decision stump, that is, a decision tree with the depth = 1. The main aim of the project is to implement all ensemble methods and make a comparison of which ensemble technique improves the performance of prediction by reducing the error.

3.1.1 Design of Averaging Methods

Averaging methods are implemented using the base algorithm as the decision tree algorithm, where the data set is sampled B times to produce B different training sets to create B different decision trees. The attribute and the threshold act as the decision condition as the root and the internal nodes and the leaves are the predicted output. In Bagging, all attributes are used to form a decision tree and predict the output label. In random forest, a set of ‘m’ attributes is used from a full set of ‘p’ attributes for each split in the decision tree. From the set of ‘m’ attributes, a decision attribute is chosen and a tree is formed, this process is repeated till a stopping condition is met. Once a tree is formed, the output label is predicted, which is the leaf of the tree. Combining together the value of output label obtained for B trees, the average of the output labels are taken together, that is, $\hat{f}_{bag}$ is calculated. This $\hat{f}_{bag}$ is applied on the test data to find the test MSE or the error.

The averaging methods such as bagging and random forest have the similar implementation of averaging the values of the output label to produce a new label $\hat{f}_{bag}$, and is applied to find the error in the prediction model. Unlike bagging, random forest considers only a few attributes to form each split in the decision tree but bagging considers all the attributes for each split in the decision tree. During each split in the decision tree, the attribute and threshold that produces the minimum training RSS (residual sum of squares) or error is considered as the decision condition for that split. Since, random forest considers less attributes to predict the label than bagging, it considers the attributes that best explains the output label and so the variance is reduced much more than bagging.

The design or the structure of the blocks in the implementation of the programs in R Programming Language is shown as a combined diagram below portraying both the averaging methods, highlighting the difference in implementation of the algorithm. The total number of trees created in both bagging and random forest is B decision trees.

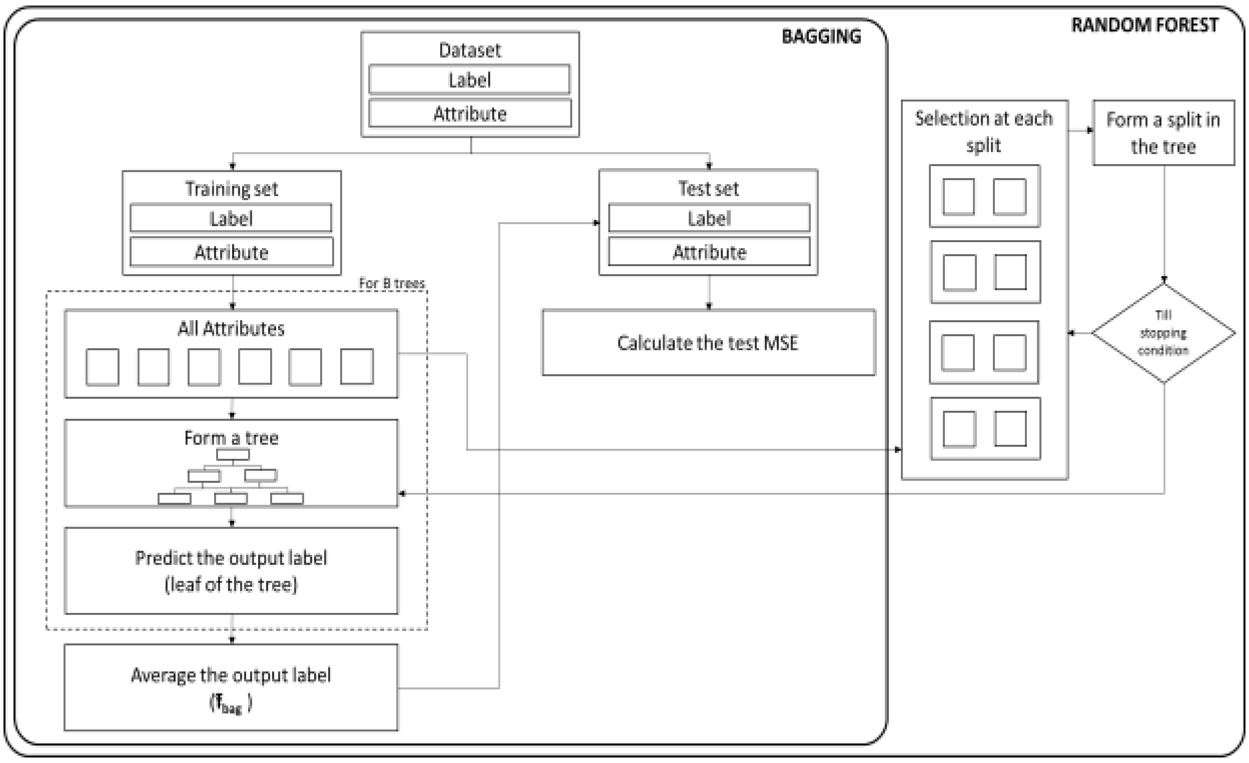


Fig 3: Design of Implementation of Averaging Methods (Bagging and Random Forest)

3.1.2 Design of Boosting Methods

Boosting methods are implemented with the base algorithm as decision trees with depth as 1, which is implemented using decision stumps. Ensemble methods work on the concept of creating a large number of simple trees, boosting method work on this concept of producing simple trees by sequentially updating the residual values, which is used in the next tree that is to be formed. Like all machine learning algorithms, boosting methods split the dataset into training set and test set, then the decision stump algorithm is applied on the training data. A few attributes are taken in boosting to form a decision stump, thus the output label is predicted. For choosing the decision stump, the attribute and the threshold that produces the minimum training RSS (residual sum of squares) or error is considered as the best decision stump for splitting the dataset and to predict the output label.

First, the residuals that are used for prediction of the label is initialised to the actual output label. After finding the best decision stump (attribute and threshold), the residuals are updated by subtracting the residual value found in last decision stump to the shrunken value of the predicted output ($\eta \hat{f}^b$), where $\eta = 0.01$ is a constant to shrink the predicted output. These residuals are used for the next iteration or for forming the next decision stump to predict the output.

By sequentially updating the values and forming decision stump improves the performance of prediction of the output label. So, the boosting method reduces the variance much more when compared to the averaging methods and thus reduces the error generated in the end or the test MSE.

The design implemented in boosting is shown below, where finally all the shrunken predicted output is added together and applied on the test set to calculate the test MSE or the error.

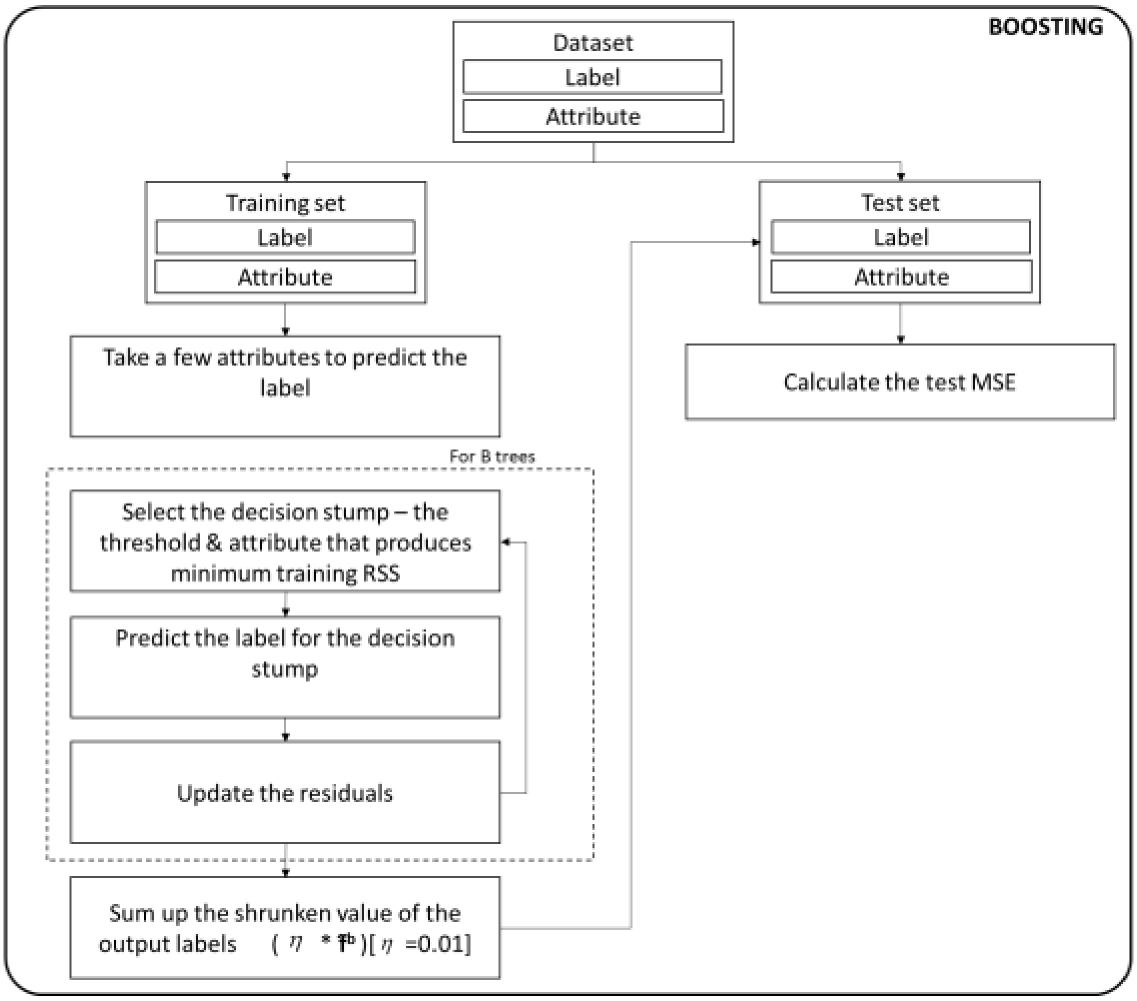

BOOSTING
Dataset
Label
Attribute
Training set
Label
Attribute
Test set
Label
Attribute
Take a few attributes to predict the label
Calculate the test MSE
For B trees
Select the decision stump – the threshold & attribute that produces minimum training RSS
Predict the label for the decision stump
Update the residuals
Sum up the shrunken value of the output labels (η * f^b)[η =0.01]

Fig 4: Design of Implementation of Boosting Methods (Boosting)

3.2 Datasets Used

The algorithms are executed and tested in all the datasets listed below. The main big datasets are SkillCraft1 Master Table dataset and Individual Household Electric Power Consumption Dataset, but for testing the values of the error, a simple built-in dataset was used, such as, the Boston dataset.

Boston Dataset: This is a built-in dataset present in R, focusing on the housing values in suburbs of Boston. It consists of 14 attributes and 506 instances. This dataset is loaded by calling the MASS library, that is, the command library(MASS) is added. Then the Boston dataset can be used for further evaluation. All the three algorithms, such as bagging, random forest and boosting is applied on the Boston dataset to check whether the values obtained.

Label : medv (median value of owner occupied homes in $1000

Attributes	Information (all are continuous)
crim	Per capita crime rate by town
zn	Proportion of residential land zoned for lots over 25,000 sqft

indus	Proportion of non-retail business acres per town
chas	Charles River dummy variable (=1 if tract bounds river, 0 otherwise)
nox	Nitrogen oxides concentration (parts per 10 million)
rm	Average number of rooms per dwelling
age	Proportion of owner-occupied units built prior to 1940
dis	Weighted mean of distances to five Boston employment centres
rad	Index of accessibility to radial highways
tax	Full-value property-tax rate per $10,000
ptratio	Pupil-Teacher ratio by town
black	Proportion of blacks by town
lstat	Lower status of the population (percent)

SkillCraft1 Master Table Dataset: This dataset is in the gaming area, is multivariate and has real and continuous numbers. It has 20 attributes with 3395 instances. The dataset has aggregated the screen movements into screen-fixations using a Salvucci & Goldberg dispersion-threshold algorithm, and defined Perception Action Cycles (PACs) as fixations with at least one action. The time is recorded in terms of timestamps in the StarCraft 2 replay file. When the game is played on 'faster', 1 real-time second is equivalent to roughly 88.5 timestamps. A list of possible game actions is discussed and the actions per minute is also recorded as an attribute.

The dataset is downloaded from the UCI repository, which has 3395 instances which is suitable for regression. The following are the input attributes used to predict the output label 'Total Hours', that is, the total hours spent playing the game.

Label: Total Hours

Attributes	**Information (all are continuous)**
GameID	Unique ID
LeagueIndex	Bronze, Silver, Gold, Platinum, Diamond, Master, GrandMaster, and Professional leagues coded 1-8
Age	Age of each player
HoursPerWeek	Reported hours spent playing per week
Actions per minute (APM)	The number of actions recorded per minute
Selection By Hot Keys	Number of unit or building selections made using hotkeys per timestamp
AssignToHotkeys	Number of units or buildings assigned to hotkeys per timestamp
UniqueHotkeys	Number of unique hotkeys used per timestamp
MinimapAttacks	Number of attack actions on minimap per timestamp
MinimapRightClicks	number of right-clicks on minimap per timestamp

NumberOfPACs	Number of PACs per timestamp
GapBetweenPACs	Mean duration in milliseconds between PACs
ActionLatency	Mean latency from the onset of a PACs to their first action in milliseconds
ActionsInPAC	Mean number of actions within each PAC
TotalMapExplored	The number of 24x24 game coordinate grids viewed by the player per timestamp
WorkersMade	Number of SCVs, drones, and probes trained per timestamp
UniqueUnitsMade	Unique unites made per timestamp
ComplexUnitsMade	Number of ghosts, infestors, and high templars trained per timestamp
ComplexAbilitiesUsed	Abilities requiring specific targeting instructions used per timestamp

Individual Household Electric Power Consumption Dataset: This dataset relates to the physical area with real and continuous values containing 9 attributes and 2075259 instances, that is, the archive contains 2075259 measurements gathered between December 2006 and November 2010 (47months). Global_active_power*1000/60 - sub_metering_1 - sub_metering_2 - sub_metering_3 represents the active energy consumed every minute (in watt hour) in the household by electrical equipment.

The dataset is downloaded from the UCI repository, which is suitable for regression. The following are the input attributes used to predict the output label 'Voltage', that is, the minute averaged voltage, which is represented in volts.

Label: Voltage

Attributes	**Information (all are continuous)**
Date	In the format dd/mm/yyyy
Time	In the format hh/mm/ss
Global_active_power	household global minute-averaged active power (in kilowatt)
Global_reactive_power:	household global minute-averaged reactive power (in kilowatt
Global_intensity	household global minute-averaged current intensity (in ampere)
sub_metering_1	Energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave
sub_metering_2	Energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.
sub_metering_3	Energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

3.3 Testing Methodologies and Procedures

In addition to the Development phase, the ensemble methods of Bagging, Random Forest and Boosting need to pass through test execution and analysis activities to ensure the desired results. The three different methods implemented in the current project have been passed through the following procedures to ensure the robustness of the final system.

3.3.1 Strategizing the Test Requirements

Before the start of testing activities, there is a definite need to completely understand all the testable requirements. These have been finalized as per the requirements and design projected during the development and the testing use cases have been defined. On completion of the use case design, a traceability matrix was created to maintain the traceability between the use cases and the test conditions or requirements and each test requirement was assigned a specific priority based on the criticality of the underlying process. This step in the testing process supports in estimating the effort required for complete testing of the three different methods of Bagging, Random Forest and Boosting.

3.3.2 Test Planning

During this phase, the roadmap for testing needs to be defined and this will specify the types of testing to be done and the exact test methodology to be followed along with the planned test schedule.

This phase also clarified the final build & deployment procedure, which is very critical in in testing. The definitions and the testing scope for incremental builds and code freeze builds were clarified during this phase.

3.3.3 Test Design

The Test Design phase primarily involves designing of the test cases (both positive and negative conditions) required to test the three ensemble methods in a comprehensive manner. Following were the set of activities involved:

- Preparation of Test cases
- Review of Test cases
- Preparation of Test Data
- Review of Test Data

3.3.4 Test Execution

During the Test Execution phase, all test cases were categorized as below:

- Priority 1 (High)
- Priority 2 (Medium)
- Priority 3 (Low)

These test cases were then executed according to the above priority categorization. The key activities in this phase would be as follows:

- Execution of all Test cases as per priority
- Defects identified during first level execution were logged in the test results section in each of the cases and tracked to closure
- Re-testing of defects from failed test cases and cases where the code defects have been fixed

3.3.5 Test Closure

On completion of testing for each method, the following were captured and logged in the test cases:

- Defect Categorization will be as:
    - Severity – Critical, High, Medium and Low
    - Status – Pending, Fixed, Complete, Deferred, Rejected and Duplicate
- Provide an overall assessment of quality of the system
- Provide recommendations for further testing/ implementations, if required

3.4 Comparison between the algorithms

All algorithms are applied on the same data, that is, on the Boston dataset, and checked the difference between the actual output and the predicted output.

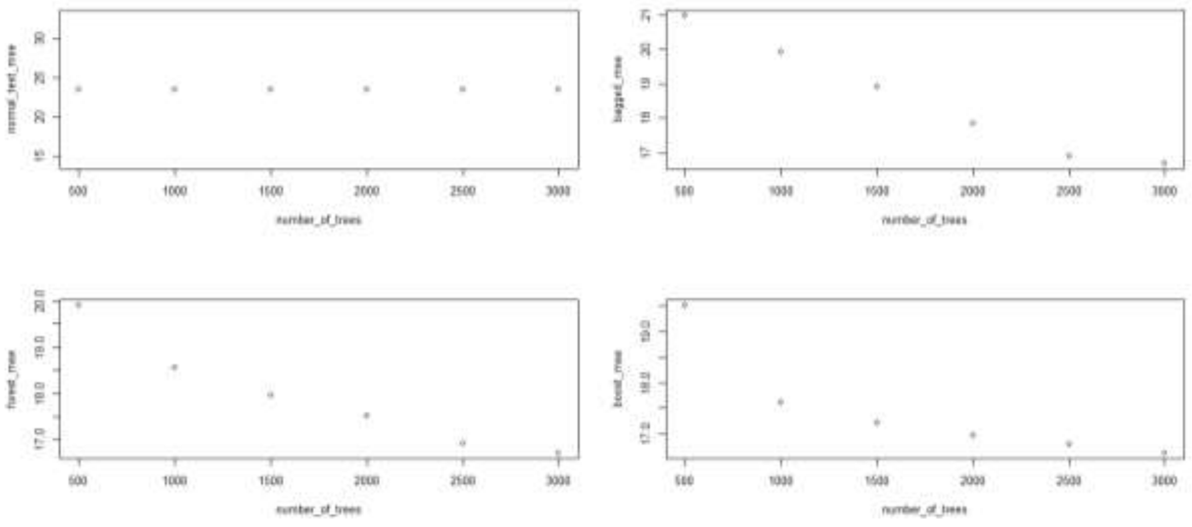


Fig 5: Comparison between graphs – number of trees vs test MSE (top-left: Implementation without ensemble methods, top-right: Bagging, bottom-left: Random forest, bottom-right: Boosting)

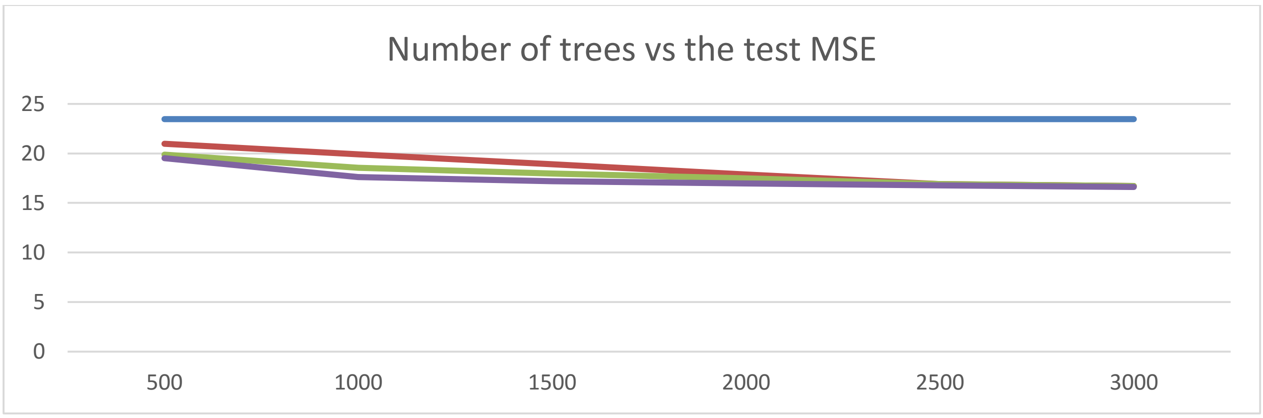


Fig 6: Comparison between graphs – number of trees vs test MSE (blue: Implementation without ensemble methods, red: Bagging, green: Random forest, purple: Boosting)

The blue line indicates the normal test MSE found, without applying any ensemble technique. The red line indicates the dependence of test MSE obtained on bagging based on the number of trees formed, the green line indicates the dependence of test MSE obtained on random forest based on the number of trees formed, the purple line indicates the dependence of test MSE obtained on boosting based on the number of trees formed.

3.4.1 Statistical Comparison

The number of times the algorithms have been correct and wrong, allows us to perform statistical analysis. This comparison was made on the Boston dataset to verify the answer, and since the total number of instances is less, that is 253.

	Correct	Wrong
Algorithm without using ensemble methods	196	57
Bagging	232	21
Random Forest	238	15
Boosting	246	7

The algorithm created without using ensemble methods has an error rate of (57/253) = 0.2253 = 22.53% and the error rate produced by bagging is (21/253) = 0.083 = 8.3%and the error rate produced by random forest is (15/253) = 0.0592 = 5.9% and the error rate produced by boosting is (7/253) = 0.0276 = 2.76%.

3.4.1 Statistical Comparison between algorithm without ensemble methods and Bagging

To statically compare the algorithm which was implemented without ensemble methods and the bagging algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 22.53% and bagging has an error rate of 8.3%, which is 14.23% lesser. To prove that this value is statistically significant, the ‘p’ value is calculated. This is done by using R function pbinom(), that is, pbinom(0,36,0.5), the ‘p’ value is 1.455192e-11, which is very less than 0.01. This proves that the value is highly statistically significant. Thus, bagging is a better algorithm than the normal implementation to predict the label.

Algorithm without ensemble methods	Bagging Algorithm	
	Correct	Wrong
Correct	196	0
Wrong	(232-196)=36	21

3.4.2 Statistical Comparison between algorithm without ensemble methods and Random Forest

To statically compare the algorithm which was implemented without ensemble methods and the random forest algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 22.53% and random forest has an error rate of 5.9%, which is 16.63% lesser. To prove that this value is statistically significant, the 'p' value is calculated. This is done by using R function pbinom(), that is, pbinom(0,50,0.5), the 'p' value is 8.881784e-16, which is very less than 0.01. This proves that the value is highly statistically significant. Thus, random forest is a better algorithm than the normal implementation to predict the label.

Algorithm without ensemble methods	Random Forest Algorithm	
	Correct	Wrong
Correct	196	0
Wrong	(238-196)=42	15

3.4.3 Statistical Comparison between algorithm without ensemble methods and Boosting

To statically compare the algorithm which was implemented without ensemble methods and the boosting algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 22.53% and random forest has an error rate of 2.76%, which is 19.77% lesser. To prove that this value is statistically significant, the 'p' value is calculated. This is done by using R function pbinom(), that is, pbinom(0,50,0.5), the 'p' value is 8.881784e-16, which is very less than 0.01. This proves that the value is highly statistically significant. Thus, boosting is a better algorithm than the normal implementation to predict the label.

Algorithm without ensemble methods	Boosting Algorithm	
	Correct	Wrong
Correct	196	0
Wrong	(246-196)=50	7

3.4.4 Statistical Comparison between Bagging and Random Forest

To statically compare the bagging algorithm and random forest algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 8.3% and random forest has an error rate of 5.9%, which is 2.4% lesser. To prove that this value is statistically significant, the 'p' value is calculated. This is done by using R function pbinom(), that is, pbinom(0,6,0.5), the 'p' value is 0.015625, which is less than 0.01. This proves that the value is highly statistically significant. Thus, random forest is a better algorithm than bagging.

Bagging	Random Forest	
	Correct	Wrong
Correct	232	0
Wrong	(238-232)=6	15

3.4.5 Statistical Comparison between Bagging and Boosting

To statically compare the bagging algorithm and boosting algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 8.3% and random forest has an error rate of 2.76%, which is 5.54% lesser. To prove that this value is statistically significant, the 'p' value is calculated. This is done by using R function pbinom(), that is, pbinom(0,14,0.5), the 'p' value is 6.103516e-05, which is less than 0.01. This proves that the value is highly statistically significant. Thus, boosting is a better algorithm than bagging.

Bagging	Boosting	
	Correct	Wrong
Correct	232	0
Wrong	(246-232) = 14	7

3.4.6 Statistical Comparison between Random Forest and Boosting

To statically compare the random forest algorithm and boosting algorithm, the error rate is compared, in which the algorithm where ensemble methods is not applied is 5.9% and random forest has an error rate of 2.76%, which is 3.14% lesser. To prove that this value is statistically significant, the 'p' value is calculated. This is done by using R function pbinom(), that is, pbinom(0,14,0.5), the 'p' value is 6.103516e-05, which is less than 0.01. This proves that the value is highly statistically significant. Thus, boosting is a better algorithm than bagging.

Random Forest	Boosting	
	Correct	Wrong
Correct	238	0
Wrong	(246-238) = 8	7

3.5 Research without applying ensemble methods

3.5.1 Aim

The main aim is to calculate the error in prediction without applying ensemble techniques

3.5.2 Design

The implementation of decision tree algorithm is applied on real world dataset to predict the output label. The error value, that is, the difference between the actual output and the predicted output is found. This error value is found after applying the prediction model on the test data.

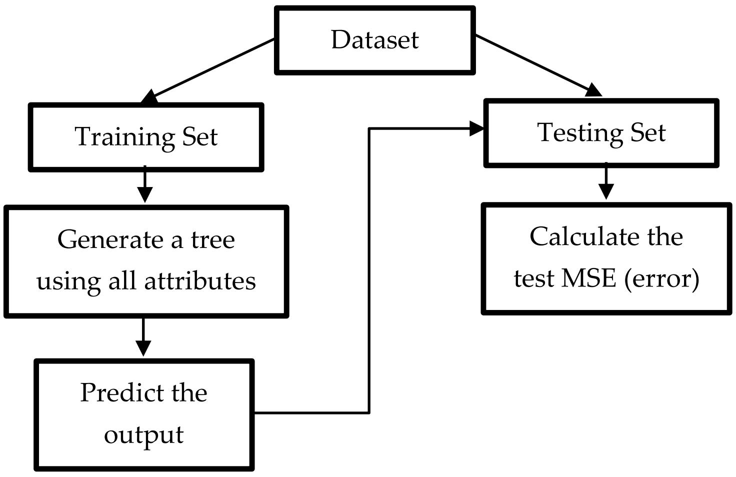

Dataset
Training Set
Testing Set
Generate a tree using all attributes
Calculate the test MSE (error)
Predict the output

Fig 7: Block Diagram of implementation of Decision tree algorithm (without applying ensemble methods)

3.5.3 Results Obtained

The results of the error in prediction is obtained by applying the model on the test data, that is, the test MSE value is found.

Test MSE value obtained is 23.47109 when applied on Boston dataset

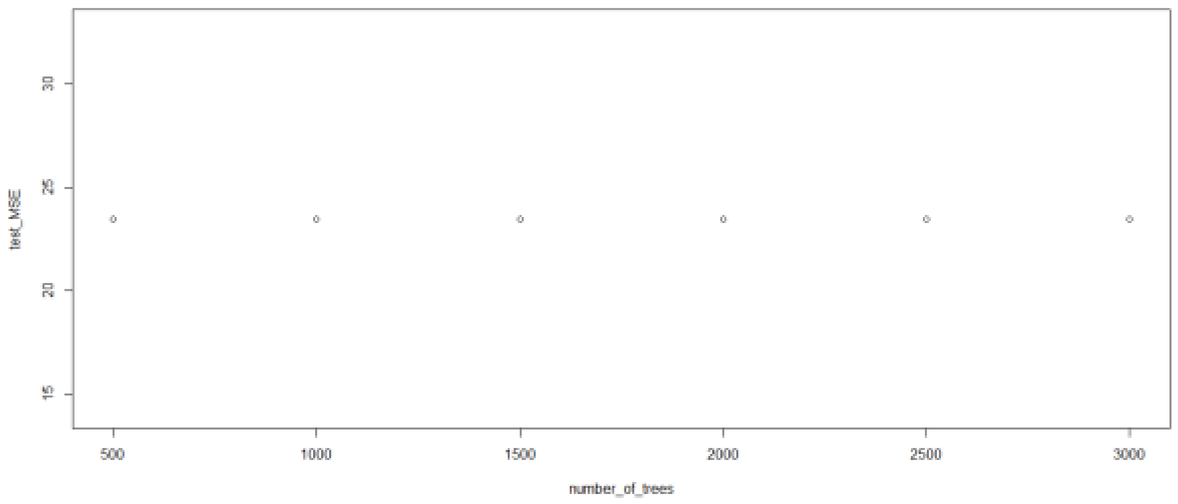


Fig 8: Graph 1 - Graph that shows the number of trees vs test MSE (without applying ensemble methods) - Boston dataset

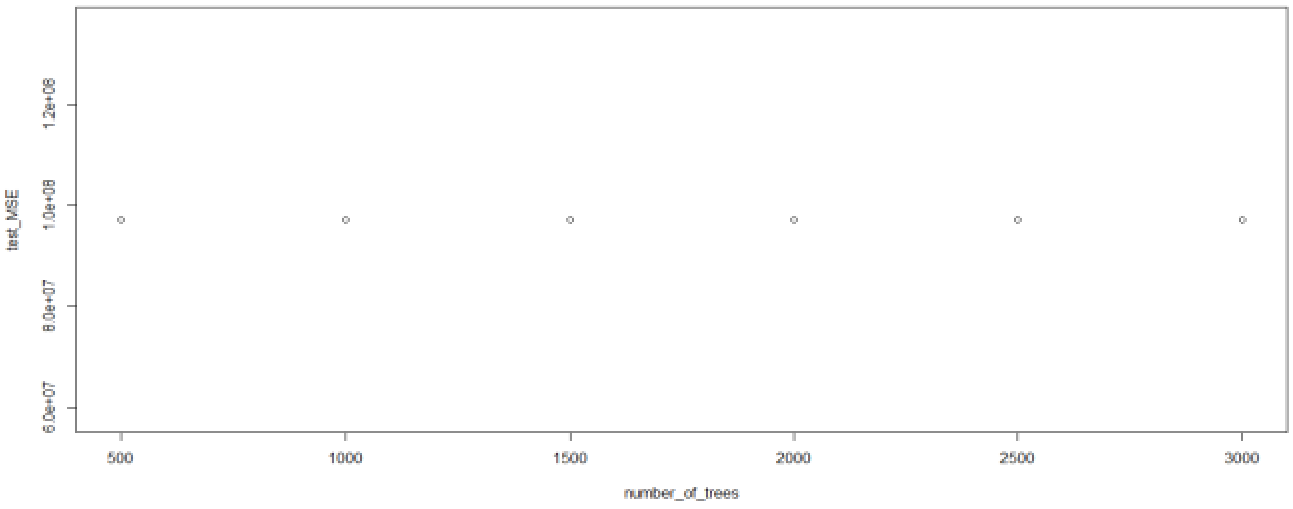


Fig 9: Graph 2 - Graph that shows the number of trees vs test MSE (without applying ensemble methods) – SkillCraft1 dataset

3.5.1 Conclusion

The test MSE value is less, showing that the base algorithm like decision tree learning reduces the difference between the actual output and predicted output. This test MSE does not depend on the number of trees formed as the test MSE is not a function of number of trees applied.

3.6 Research in Bagging

3.6.1 Aim

The goal of bagging is to reduce the variance by combining different training samples together by creating m different samples of size n, by drawing n examples from the original data, with replacement and averaging all the samples will reduce the variance of the model and to avoid over-fitted model.

- Calculate test MSE
- Plotting the error and the number of trees
- Spotting visual correlations between errors

3.6.2 Knowledge Acquisition

The first step is to split the training data and the testing data in the dataset and then apply the decision tree algorithm on training data. The training data is sampled every time a tree is formed using the all the attributes in the dataset. Using the decision tree, the label of the dataset is predicted and is compared with the actual output that is to be obtained.

Choice of Programming Language: R Programming Language is the preferred language because the bagging program code already exists in this language. So, the bagging function in R can be used to test our program and check the test MSE value obtained with the value obtained using the function.

Visualisation: After running the program for different number of trees, the test MSE value is calculated. It was observed that as the number of trees increase, the test MSE (or error) value decreases. This is because as we are dividing the values of the output obtained by forming each tree by the total number of trees, or taking the average of the label values with respect to the number of trees formed.

3.6.3 Design

Bagging is an averaging method, so a sample training set is taken for forming each tree. While forming each tree, a decision tree is formed with respect to all the attributes in the dataset to predict the output label. Each time a split in the tree is considered, the best attribute and threshold, that is, the attribute that produces minimum training RSS with a particular threshold value is considered as the best split at that point in the tree. Similarly at each split of the tree is made to form a big decision tree, and the output is predicted, which is the leaf of the tree. After forming B different decision trees, the predicted outputs are combined together and the average of the predicted outs are found ($\hat{f}_{bag}$). This averaged predicted output ($\hat{f}_{bag}$) is applied on the test data set to calculate the test MSE value or the error in the prediction.

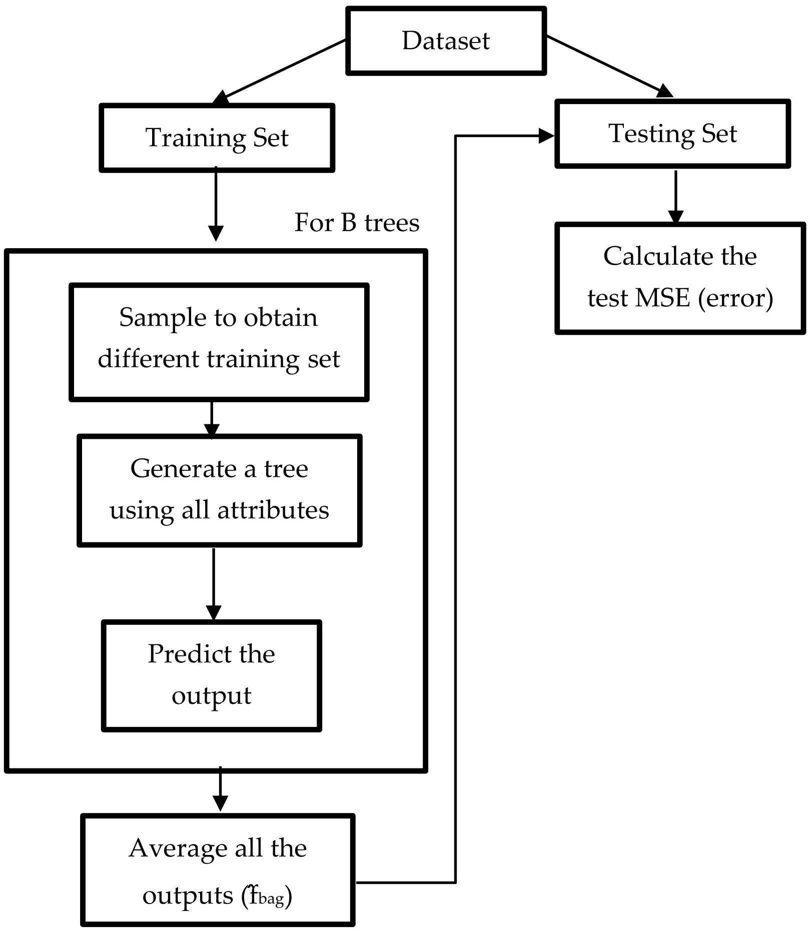

Dataset
Training Set
Testing Set
For B trees
Calculate the test MSE (error)
Sample to obtain different training set
Generate a tree using all attributes
Predict the output
Average all the outputs ($\hat{f}_{bag}$)

Fig 10: Block Diagram of implementation of Bagging

3.6.4 Results Obtained

The implementation of bagging is applied on the SkillCraft1 dataset to observe the pattern in which the error or the test MSE reduces with respect to the number of trees generated to predict the final output. When decisions of a group of people is combined to produce a final output, the decision seems to be better and the output becomes from better to best when the number of people in the group increases to share the ideas. Similarly, bagging combines all the output generated while forming each tree and averaged value becomes the most effective, such that the error or the test MSE value is decreased.

The graph shows below is a proof of this concept, which portrays that as the number of trees generated to predict the final output, the test MSE value decreases. Slight deviation in the graph is observed while averaging the predictions, but it clearly shows that the error value decreases with the increase in number of trees. The test MSE depends on the bias and variance of the output predicted in terms of the input attributes. Since the main aim of bagging is to reduce the variance or overcome the over-fitted problem, the number of trees is increased.

Variance ∝ (1 / number of trees)

The test MSE consists of an irreducible error (Variance of a constant) and a reducible error. The reducible error is reduced using bagging and thus the test MSE is finally reduced.

The values obtained are the following in Boston dataset:

Number of trees	Test MSE
500	20.96287
1000	19.9108
1500	18.90652
2000	17.86202
2500	16.91347
3000	16.70934

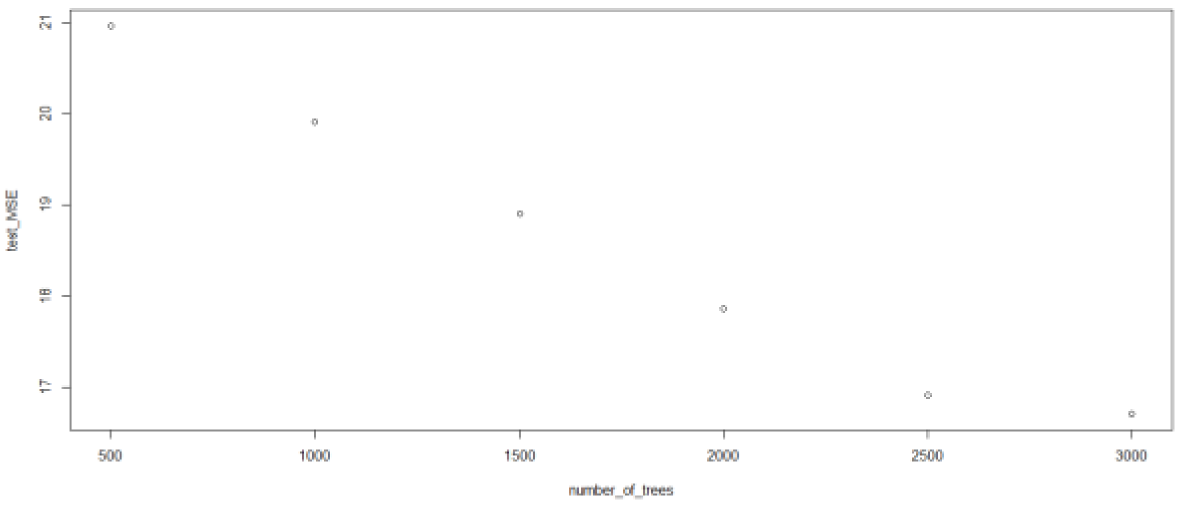


Fig 11: Graph 3- Number of trees formed in Bagging vs the test MSE obtained (Boston dataset)

The values obtained are the following in SkillCraft1 dataset:

Number of trees	Test MSE
500	515559998
1000	513094415
1500	510615606
2000	509589345
2500	509022619
3000	508088563

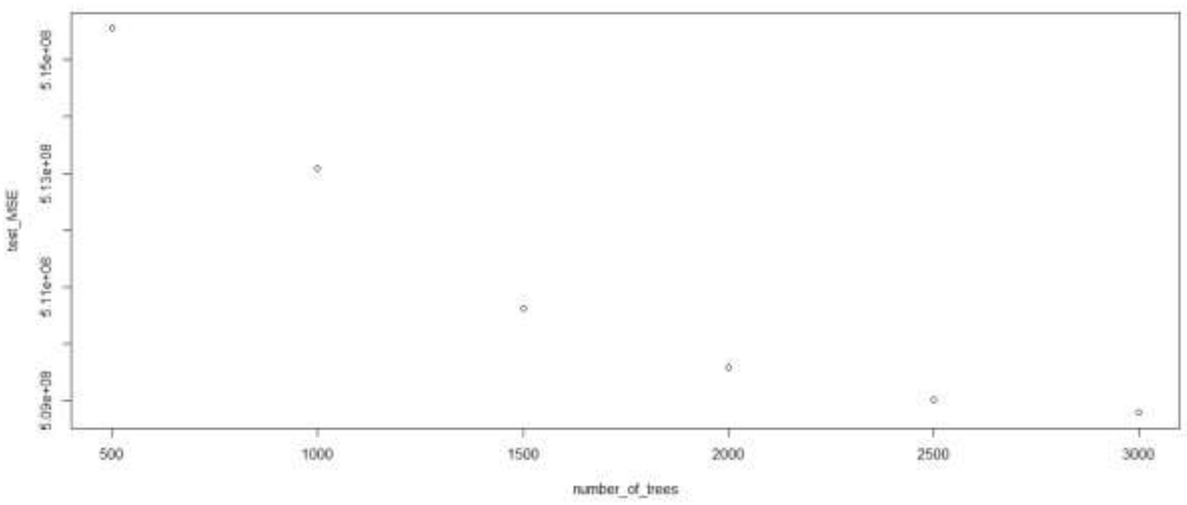


Fig 12: Graph 4- Number of trees formed in Bagging vs the test MSE obtained (SkillCraft1 dataset)

3.6.5 Testing

The following are the test cases done on the data to check the execution of the program:

Test Case ID: BG001

Test Heading: Test the predicted output

Test Objective: The output is predicted for each decision tree

Priority: High

Pre-condition:

- All attributes are taken as input to form a decision tree
- Training data is formed by sampling the dataset
- Tree is formed to predict the output label
- Create an empty matrix for storing the predicted output

Test Steps: Check whether the predicted output is formed

Test Data: The attribute to be tested is given as the input. Let the attribute be x=1, 2, 3, 4, and let the actual output be y = 2, 4, 6, 8, and the output label is predicted.

Expected Result:

```
fhat

    [1] [2] [3] [4] [5]

[1] 5    5    5   5    5
```

Actual Result:

```
fhat

    [1] [2] [3] [4] [5]

[1] 5    5    5   5    5
```

PASS/ FAIL: PASS
Remarks/ Notes: The predicted output is named as fhat.

Test Case ID: BG002

Test Heading: Test the bagged output

Test Objective: The bagged output is tested as whether it is the average of all the outputs generated for each decision tree.

Priority: High

Pre-condition: All the output are formed to find the bagged output value

Test Steps:

- The predicted output for each tree is taken as the input
- The bagged output label is found

Test Data: The bagged output to be tested is given an input as the predicted output formed for each iteration. Let the predicted output be fhat=2, 4, 6, 8, and let the bagged output be fbag.

Expected Result: fbag

```
[1] 5
```

Actual Result: fbag

```
[1] 5
```

PASS/ FAIL: PASS

Remarks/ Notes: The bagged output is named as fbag.

3.6.6 Conclusion

The research found out by bagging, when we combine all the predictions obtained while formed many trees, the error in prediction is much reduced than considering a single tree algorithm for prediction.

3.7 Research in Random Forest

3.7.1 Aim

The goal of random forest is to reduce the variance by combining different training samples together. While forming a decision tree from the training samples, at each split taken in the tree, a set of 'm' attributes is chosen from full set of 'p' attributes and averaging all the values will reduce the variance of the model and to avoid over-fitted model.

3.7.2 Knowledge Acquisition

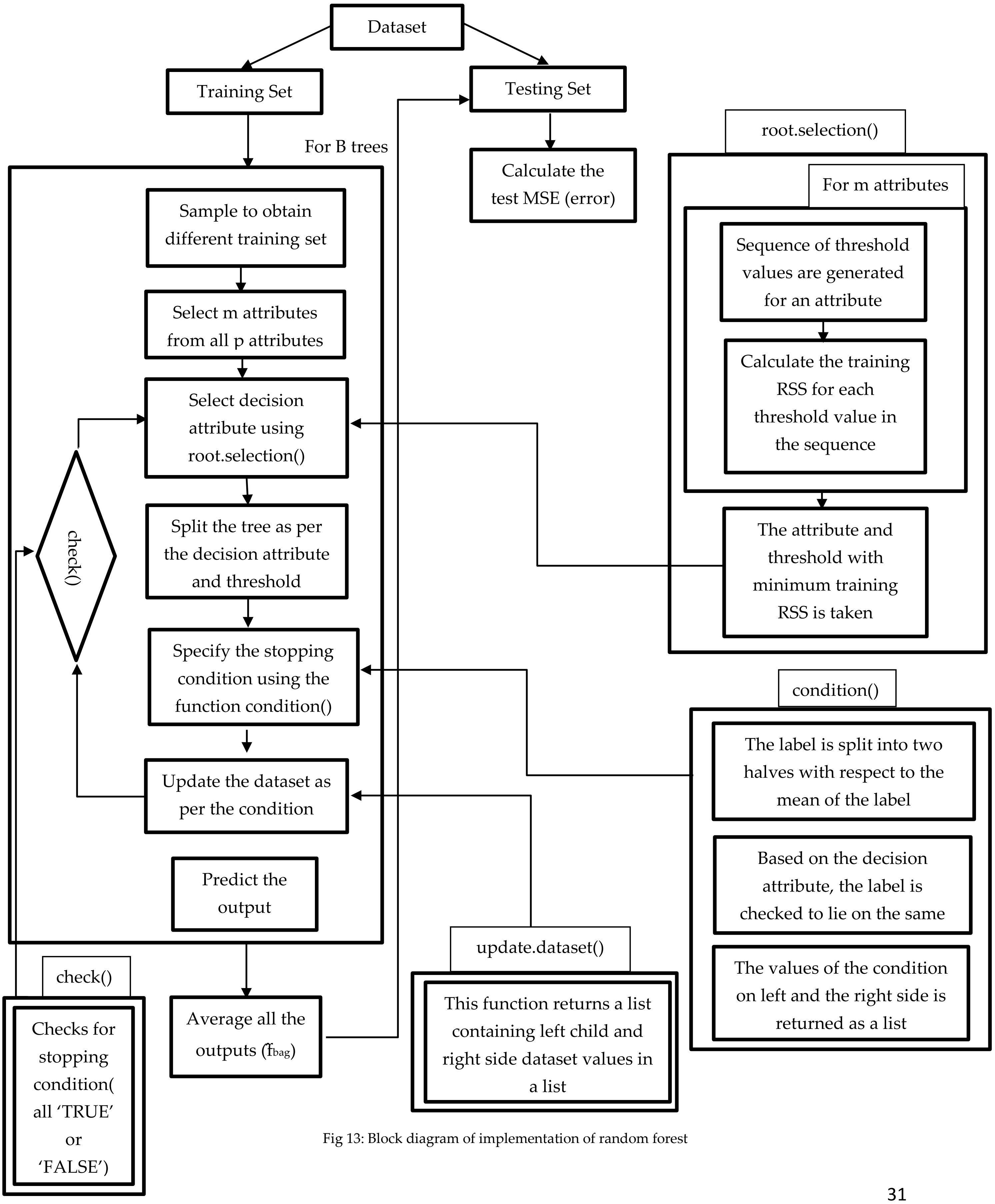
The first step is to split the training data and the testing data in the dataset and then apply the decision tree algorithm on training data. The training data is sampled every time a tree is formed using 'm' attributes among all the 'p' attributes of the dataset. Using the decision tree, the label of the dataset is predicted and is compared with the actual output that is to be obtained.

Choice of Programming Language: R Programming Language is the preferred language because the random forest program code already exists in this language. So, the function for random forest in R can be used to test our program and check the test MSE value obtained with the value obtained using the function.

Visualisation: After running the program for different number of trees, the test MSE value is calculated. It was observed that as the number of trees increase, the test MSE (or error) value decreases. This is because as we are dividing the values of the output obtained by forming each tree by the total number of trees, or taking the average of the label values with respect to the number of trees formed. This error value obtained is lesser than the value obtained in bagging. Thus the error is reduced when the number of attributes used for prediction is less.

3.7.3 Design

Random Forest is implemented in a similar manner as bagging but a different set of attributes is chosen from the full set of attributes available in the dataset. The program on random forest produces a sample of training set B times to form B decision trees. We have to select a decision attribute and threshold at each split, this is done using the function root.selection(). The root.selection() function selects an attribute, and an attribute is checked with a sequence of threshold, and the attribute with the best threshold that produces the minimum training RSS is chosen. Split the tree based on a condition, this is done using the function condition() and update the dataset as per the condition. The condition() function splits the output label into two halves. Then the stopping condition of the decision tree is decided based on whether the decision attribute chosen to split the dataset categorises the output label such that each value in a split exactly belongs to the same half of the output label. This evaluation is done using the check() function. Update the dataset until the stopping condition is satisfied. Predict the output label as per the tree formed, where the leaves of the tree is the label. Average all the output label obtained at each tree ($\hat{f}_{bag}$). Apply this averaged predicted output ($\hat{f}_{bag}$) on the test dataset to obtain the test MSE or the error



Dataset
Training Set
Testing Set
Calculate the test MSE (error)
For B trees
Sample to obtain different training set
Select m attributes from all p attributes
Select decision attribute using root.selection()
Split the tree as per the decision attribute and threshold
Specify the stopping condition using the function condition()
Update the dataset as per the condition
Predict the output
check()
Average all the outputs ($\hat{f}_{bag}$)
check()
Checks for stopping condition(all 'TRUE' or 'FALSE')
root.selection()
For m attributes
Sequence of threshold values are generated for an attribute
Calculate the training RSS for each threshold value in the sequence
The attribute and threshold with minimum training RSS is taken
condition()
The label is split into two halves with respect to the mean of the label
Based on the decision attribute, the label is checked to lie on the same
The values of the condition on left and the right side is returned as a list
update.dataset()
This function returns a list containing left child and right side values in a list

Fig 13: Block diagram of implementation of random forest

3.7.4 Results Obtained

The variance of the data is further reduced using random forest. Like bagging, random forest also forms B different decision trees and an average of all the predicted output is taken as the final prediction of the model. While forming a decision, at each split in the tree, a set of attributes 'm' is chosen from a full set of attributes 'p', from which we select a decision attribute to form the tree. By selecting a set of attributes from a full set of attributes available at each split in decision tree, random forest decreases the error or the difference between the prediction and the actual output.

The values obtained are the following in Boston dataset:

Number of trees	Test MSE
500	19.89254
1000	18.55201
1500	17.96306
2000	17.5034
2500	16.90784
3000	16.70824

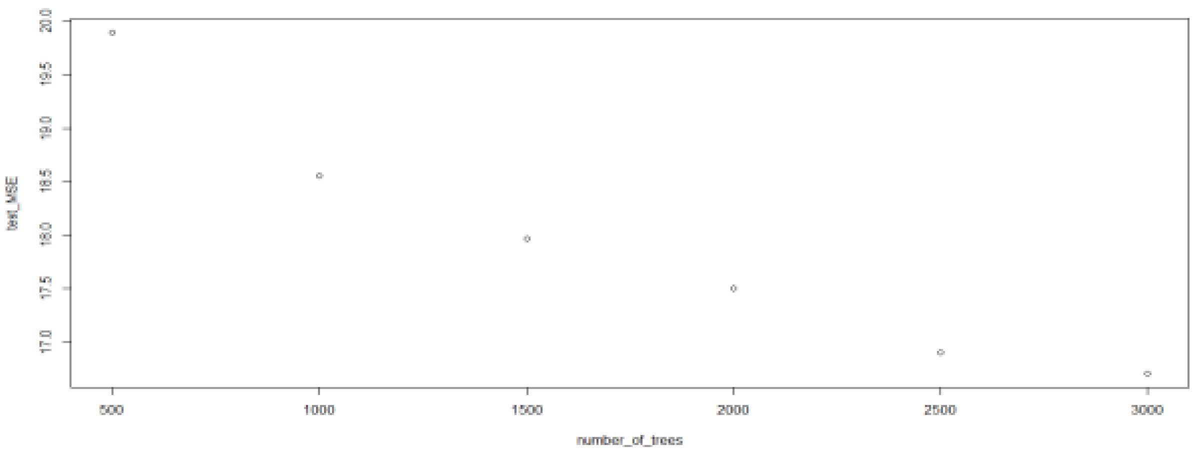


Fig 14: Graph 4- Number of trees formed in Random Forest vs the test MSE obtained(Boston dataset)

The values obtained are the following in SkillCraft1 dataset:

Number of trees	Test MSE
500	509714065
1000	509569573
1500	509543897
2000	509421375
2500	509220182
3000	509032276

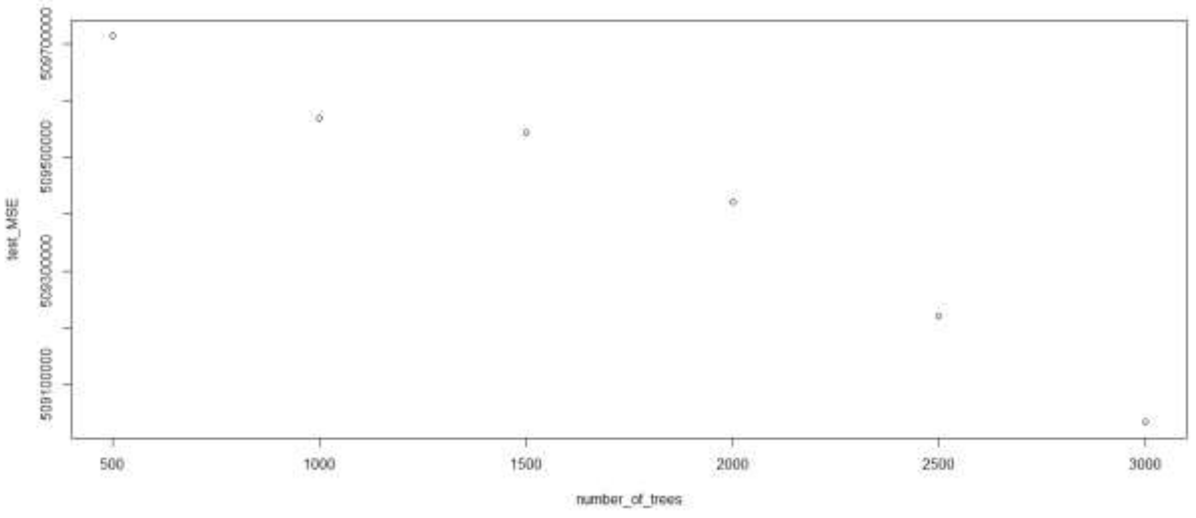


Fig 15: Graph 5- Number of trees formed in Random Forest vs the test MSE obtained (SkillCraft1 dataset)

3.7.5 Testing

The following are the test cases done on the data to check the execution of the program:

Test ID: RF001

Test Heading: Selecting an attribute for producing the decision condition

Test Objective: Checking whether the root for a particular split in the decision tree is selected

Priority: High

Pre-condition:

- The attributes among which the best attribute to be chosen is passed as into the function root.selection()
- The total number of attributes (p) present in the dataset is an input to the function
- The set of attributes to be selected (m) from the total set of attributes (p) is an input to the function
- The actual output label (y) is also given as input to the function

Test Steps:

- Give the attributes from which you have to select the best attribute as a decision condition
- The total number of attributes ‘p’ and the set of attributes to be chosen ‘m’ is also tested
- The function is checked whether it returns the best attribute, which produces the minimum training RSS with a particular threshold is chosen, and the threshold, and the prediction produced on the left and right side based on the decision.

Test Data: The dataset, in which the best attribute to be chosen is tested. Let the attributes in the dataset be x1=1, 2, 3, 4, and x2 = 1.5, 2.5, 3.5, 4.5, and the best attribute among the two is considered.

Let the dataset have a total number of p = 5 attributes and let us consider two attributes m = 2, from which we have to select the best attribute. The two attributes chosen are x1 and x2 and the best attribute among x1 and x2 is selected. Let the actual output y=2, 4, 6, 8.

Calculation:

Training RSS for the attribute x1:

- The left and the right prediction, that is, yless and ymore are found
    - Condition - x<s[t], where s[t] = 2.5
    - Values true for the condition – 1, 2
    - Values false for the condition – 3, 4
    - Value of yless inside the loop – 2+4 = 6
    - Value of ymore inside the loop – 6+8 = 14
    - Value of yless (outside the loop) – 6/2 = 3
    - Value of ymore (outside the loop) – 14/2 = 7
- The training RSS is calculated
    - Condition - x<s[t], where s[t] = 2.5
    - The rss value on the left side – (yless – y)^2 = [((3 – 2)^2) + ((4-3)^2)] = 1 + 1 = 2
    - The rss value on the right side – (ymore – y)^2 = [((7 – 6)^2) + ((7-8)^2)] = 1+1 = 2
    - Total training RSS = 4

Training RSS for the attribute x2:

- The left and the right prediction, that is, yless and ymore are found
    - Condition - x<s[t], where s[t] = 2.5
    - Values true for the condition – 1.5
    - Values false for the condition – 2.5, 3.5, 4.5
    - Value of yless inside the loop – 2
    - Value of ymore inside the loop – 4 + 6 + 8 = 18
    - Value of yless (outside the loop) – 2/1 =2
    - Value of ymore (outside the loop) – 18/3 =6
- The training RSS is calculated
    - Condition - x<s[t], where s[t] = 2.5
    - The rss value on the left side – (yless – y)^2 = (2 – 2)^2 = 0
    - The rss value on the right side – (ymore – y)^2 = [((6 – 4)^2) + ((6-6)^2) + ((6-8)^2)] = [4 + 0 + 4] = 8

    - Total training RSS = 8

- The attribute with the minimum training RSS is chosen, so attribute x1 is chosen as the decision attribute with the threshold 2.5. Let us consider that x1 is the first attribute among the attribute list. So, the attribute number is returned with the threshold value, the left and right prediction

Expected Result:

```
x <- attribute.selection(data, 5, 2, y)

x

[1] 1.00 2.50 3.00 7.00
```

Actual Result:

```
x

[1] 1.00 2.50 3.00 7.00
```

PASS/ FAIL: PASS

Remarks/ Notes:

The input for the function consists of the following:

- The dataset is named as 'data'
- Total attributes = p = 5
- Chosen number of attributes = m = 2
- Actual output = y

The output for the function is a vector, which consists of the following values:

- First value is the attribute number among the attribute list in the dataset
- The threshold value, through which the minimum training RSS is obtained
- The predicted value obtained on the left hand side after applying the condition using the attribute and the threshold
- The predicted value obtained on the right hand side after applying the condition using the attribute and the threshold

Test ID: RF002

Test Heading: Updating the dataset as per the condition

Test Objective: After splitting the dataset based on the condition, the dataset has to be updated to select an internal node in the tree and form a tree

Priority: High

Pre-condition:

- Based on the condition laid by the attribute and the threshold at each split in the tree, the dataset is split.
- After the dataset is split, the left child is updated first to further split and then the right child is updated.

Test Steps:

- The condition for splitting the attribute is passed for splitting the dataset based on the condition

- The split happens based on the attribute and thus the dataset is split based on that condition

- To check whether the dataset is updated

Test Data: The dataset consists of two attributes x1 and x2 to be tested is given as the input, this is to be updated after applying the condition. Let the attribute values be x1=1, 2, 3, 4, and x2=1.5, 2.5, 3.5, 4.5, and the threshold for splitting is s[t] = 2.5

Calculation:

- Condition x<s[t], where s[t] =2.5

- The dataset is updated based on the attribute x1 and the dataset, the attributes are split to produce a new dataset

- Values true to the condition on x1 – 1,2

- Values false to the condition on x1 – 3,4

- Dataset is updated as per this condition

    - Values of x1- 1, 2 and x2 - 1.5, 2.5 on the left split

    - Values of x1 – 3,4 and x2 – 3.5, 4.5 on the right split

- Both the splits are updated on the on a list, the list should be the return value of the function

Expected Result: dataset_list

```
[[1]]

x1   x2

1     1.5

2     2.5

[[2]]

x1  x2

3    3.5

4    4.5
```

Actual Result: dataset_list

```
[[1]]

x1    x2

1      1.5

2      2.5

[[2]]

x1    x2

3     3.5

4     4.5
```

PASS/ FAIL: PASS

Remarks/ Notes: All the attributes in the dataset are split based on the decision condition selected from an attribute.

3.7.6 Conclusion

The variance of the data is decreased by increasing the number of trees formed. This variance is reduced in random forest than bagging as we select a set of attributes from all the attributes. The number of attributes from which we form the tree is less and thus the variance is decreased.

3.8 Research in Boosting

3.8.1 Aim

The goal of boosting is to compute a sequence of simple trees, where each successive tree is built for the prediction residuals of the preceding tree. At each step of the boosting, a best partitioning of the data is determined, and the deviations of the observed values from the respective means or the residuals for each partition are computed and updated at each iteration and boosting tries to prevent the situation of over-fitting because it fits the model slowly.

3.8.2 Knowledge Acquisition

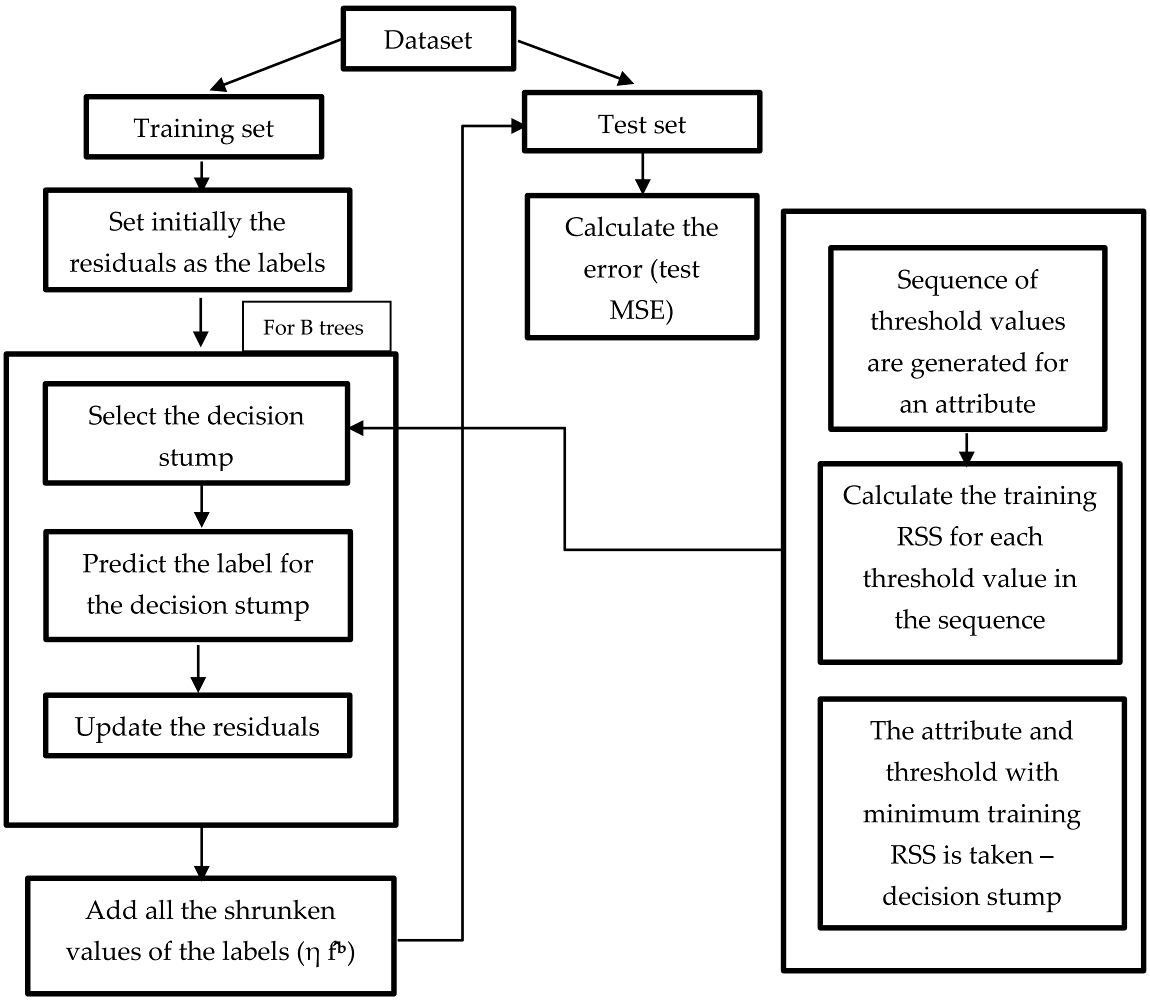
The first step is to split the training data and the testing data in the dataset and then apply the decision stump on training data. Using the decision stump formed, the label of the dataset is predicted and is compared with the actual output that is to be obtained.

Choice of Programming Language: R Programming Language is the preferred language because the random forest program code already exists in this language. So, the function for boosting in R can be used to test our program and check the test MSE value obtained with the value obtained using the function.

Visualisation: After running the program for different number of trees, the test MSE value is calculated. It was observed that as the number of trees increase, the test MSE (or error) value decreases. This is because as we are dividing the values of the output obtained by forming each tree by the total number of trees, or taking the average of the label values with respect to the number of trees formed. This error value obtained is lesser than the value obtained in bagging as well as random forest. Thus the error is reduced in boosting methods than averaging methods, so when the values are sequentially updated, the variance is even more reduced number of attributes used for prediction is less.

3.8.3 Design

Boosting is different from the averaging methods, where we sequentially update the residuals to predict the label. First, the residuals are initialised as the output label and the function is set to predict the label. Select the decision stump for splitting the dataset and predicting the label from attribute list and sequence of threshold values and predict the output label. Then the residual is updated by subtracting a shrunken value of the predicted output. Add all the shrunken values of the predicted output ($\eta \hat{f}^b$) and apply the test set to calculate the test MSE or the error.



Dataset
Training set
Test set
Set initially the residuals as the labels
Calculate the error (test MSE)
For B trees
Select the decision stump
Predict the label for the decision stump
Update the residuals
Add all the shrunken values of the labels (η f̂ᵇ)
Sequence of threshold values are generated for an attribute
Calculate the training RSS for each threshold value in the sequence
The attribute and threshold with minimum training RSS is taken – decision stump

Fig 16: Block diagram of implementation of Boosting

Boosting allows the data to be sequentially updated after finding the decision stump, that is, the attribute and the threshold value through which we can split the data. The residual values are updated after forming each decision stump, this helps in reducing the variance further and the error in the data, the test MSE is lesser than the values obtained in bagging and random forest.

The values obtained are the following in Boston dataset:

Number of trees	Test MSE
500	19.51232
1000	17.60388
1500	17.20956
2000	16.96985
2500	16.78461
3000	16.62095

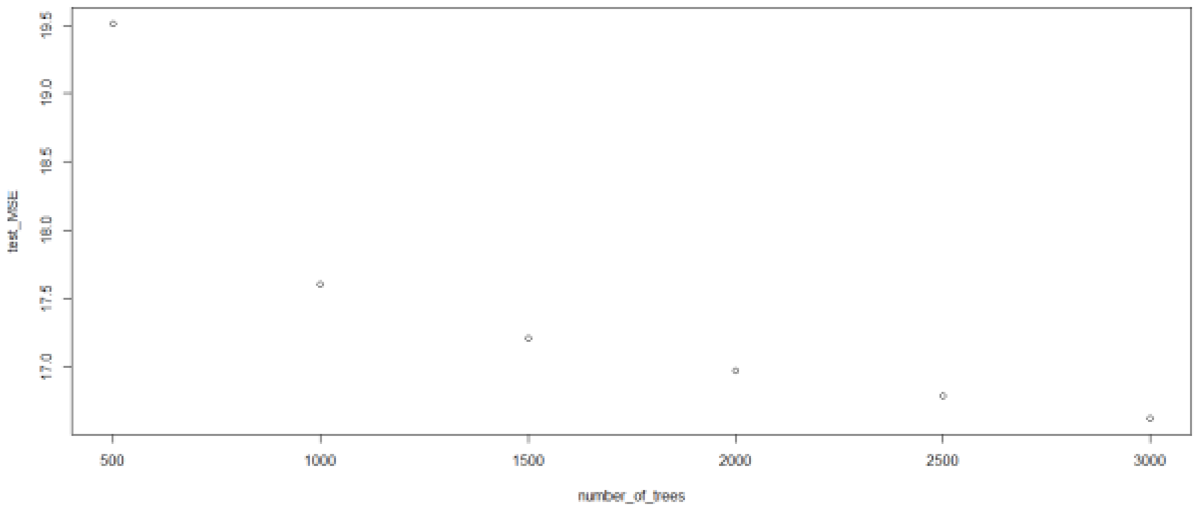


Fig 17: Graph 3- Number of trees formed in Boosting vs the test MSE obtained (Boston dataset)

The values obtained are the following in SkillCraft1 dataset:

Number of trees	Test MSE
500	507955075
1000	507869870
1500	507859656
2000	507817260
2500	507800166
3000	507757771

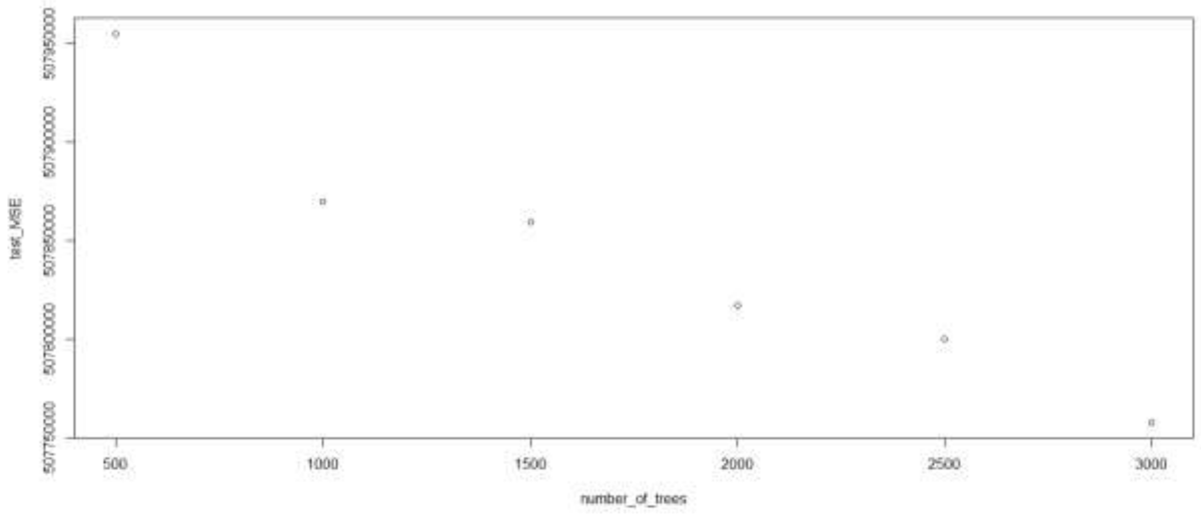


Fig 18: Graph 4- Number of trees formed in Boosting vs the test MSE obtained (SkillCraft1 dataset)

3.8.5 Testing

The following are the test cases done on the data to check the execution of the program:

Test ID: BT001

Test Heading: Checking range matrix

Test Objective: Forming the range matrix with maximum and minimum values of the attribute chosen

Priority: Medium

Pre-condition:

- The attribute values must be initialised with the values in the training dataset.
- Form an empty matrix and name it as 'range' to store the values.

Test Steps:

- Give the attribute value as input to calculate the range of the matrix
- Print the range matrix after execution
- Check whether the maximum and the minimum values of the matrix are rounded up and displayed in the range matrix
- The first column should have the maximum value and the second column should have the minimum value of the attribute chosen.

Test Data:

The attribute to be tested is given as the input. Let the attribute be x=1, 2, 3, 4, these values are passed for testing the range matrix

Calculation:

- Maximum value among the attribute values – 4
- Minimum value among the attribute values – 1

Expected Result:

```
range

   [1] [2]

[1] 4   1
```

Actual Result:

```
 [1] [2]
[1] 4   1
```

PASS/ FAIL: PASS

Remarks/ Notes: The first column has the maximum value and the second column has the minimum value of the attribute ‘x’.

Test ID: BT002

Test Heading: Finding the prediction based on the split produced by the decision stump

Test Objective: Calculating the predicted value of the output

Priority: High

Pre-condition:

- A sequence of threshold values must be produced for each attribute
- Each threshold value in the sequence must give a condition on the attribute
- This condition is used as a decision stump to predict the label
- The residual required for prediction is given as input
- The counter to be checked for incrementing while generating the number of predicted values based on the condition formed by the decision stump

Test Steps:

- Give the attribute value as input
- Test the condition involving the attribute and threshold
- Check whether the sum of all predicted values is formed from the residuals based on the condition laid by the threshold and attribute, that is, the decision stump.
- Check the counter incremented while prediction occurs
- Check the values of the predicted output formed

Test Data: The attribute to be tested and the threshold value is given as the input. Let the attribute be x=1, 2, 3, 4, and the threshold value s[t] = 2.5(‘s’ is a sequence of threshold values, a particular threshold value is chosen for each iteration). The residual is used for predicting the output is also taken as the input, let the residuals be r = 2,4,6,8, and let the counters l and m be initialised to 1. The predicted output label to be tested is yless and ymore formed based on the condition.

Calculation:

- Condition - x<s[t]
- Values true for the condition – 1, 2

- Values false for the condition – 3, 4
- Value of yless inside the loop – 2+4 = 6
- Value of ymore inside the loop – 6+8 = 14
- Value of the counter l – 2
- Value of the counter m – 2
- Value of yless (outside the loop) – 3
- Value of ymore (outside the loop) - 7

Expected Result:

```
yless

[1] 3

ymore

[1] 7
```

Actual Result:

```
  yless

[1] 3

ymore

[1] 7
```

PASS/ FAIL: PASS

Remarks/ Notes: yless and ymore are the predicted values of the output label formed based on the condition given by the decision stump.

Test ID: BT003

Test Heading: Calculate the training RSS

Test Objective: To find the minimum training RSS

Priority: High

Pre-condition:

- The residuals must be updated and formula can be applied to find the training RSS
- The minimum value of the residuals is found and selected as the decision stump for splitting

Test Steps:

▪ Give residuals as the input and the predicted output obtained to calculate the residuals

▪ Apply the formula to find the training RSS based on the condition produced by decision stump

▪ Check the minimum value of the training RSS and update the attribute values as the best attribute

Test Data: The attribute to be tested is given as the input. Let the attribute be x=1, 2, 3, 4, and the threshold s[t] = 2.5, and the residuals as r = 2,4,6,8 and yless = 3 and ymore = 7, these values are passed for calculation of training RSS. The minimum value of the training RSS is checked and chosen as the best decision stump.

Calculation:

- Condition – x < s[t]
- Residual values true for the condition – 2, 4
- Residual values false for the condition – 6, 8
- Training RSS value true to the condition - (yless – r)^2 = [((3-2)^2) = 1, ((3-4)^2) = 1]
- Sum of training RSS – 2
- Training RSS value false to the condition – (ymore – r)^2 = [((7-6)^2) = 1, ((7-8)^2) = 1]
- Sum of training RSS – 2
- Total sum of training RSS – 2+2 = 4

Expected Result:

```
rss

[1] 4
```

Actual Result:

```
  rss

[1] 4
```

PASS/ FAIL: PASS

Remarks/ Notes: The value with minimum training RSS is chosen as the best decision stump

3.8.6 Conclusion

Boosting technique sequentially updates the values of the residuals, this reduces the variance than bagging and random forest. This explains boosting methods are more effective than averaging methods.

4 Self Assessment

Self Assessment involves SWOT Analysis of my work

4.1 Strengths:

Choice of R Programming: R has enhanced the implementation of the project, because it has enhanced functions and support for importing large datasets and also help in applying the underlying algorithms

4.2 Weakness:

Inexperience in certain areas: Since it is the first time implementing large datasets, it was tough to predict the answer or the error (test MSE), that will be the output of the program. So, standard functions of the R for bagging, random forest and boosting was used on the same dataset to check for the output and also underlying programs were applied as suggested by my supervisor.

4.3 Opportunities:

Working with different datasets: Different ensemble techniques like bagging, random forest and boosting was applied on different real world datasets. By working with different datasets, I was able to understand that on which dataset ensemble methods can be applied. The different datasets were found from UCI Repository

Learning R Programming: This project gave me an opportunity to learn different functionalities of R, which was not tried in the lab sessions during our course.

Learning the way of implementation of the algorithms: Even though the theory of ensemble methods was taught during the course, the practical implementation of the programs and the step by step procedure was made clear by practically implementing the programs in the project

4.4 Threats:

Time Constraints: Since the project was given a short period of time to finish, future enhancement like applying the same scenario in case of classification couldn't be done. The comparison based on regression on different real world datasets was possible

5 Professional Issues

5.1 Issues with Loading Large Dataset

Since this project deals with working on different datasets, the professional issue is concerned with loading or importing large datasets into R programming. Even though R supports large datasets, a few of the datasets are too big to be imported. This does not allow the programmer to load or read the file. There is a common error shown while forming a decision tree with large datasets. So, we are forced to take a subset of the dataset to perform further evaluation.

5.2 Ignoring the Deviations in Visualisation

When the pattern of change of the test MSE is analysed on the basis of the number of trees, the data that is favourable or belong to a certain pattern is chosen ignoring the deviation of a certain points in the graph. For example, the graph below shows a similar situation, where the deviation of the increase in test MSE for 1500 trees from the test MSE for 1000 trees, this increase is ignored and we directly consider that test MSE decreases with the increase in number of trees.

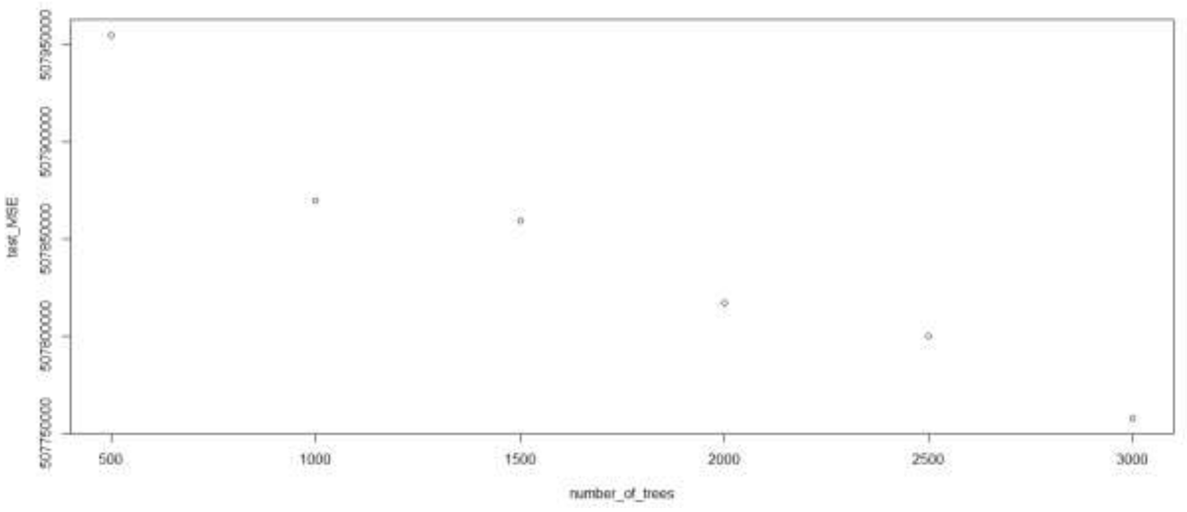


Fig 19: Graph 5- Number of trees formed in Boosting vs the test MSE obtained (SkillCraft1 dataset)

6 Conclusion

Machine learning algorithms aim at decreasing the error, that is, the difference or the gap between the actual output and the predicted output. Each algorithm tries to reduce this difference, ensemble methods are one among the machine learning algorithms which try to reduce the difference factor. Thus, ensemble methods reduces the error drastically and is also simple and effective because only easy algorithms are implemented many times to predict a better output, closer to the actual value.

References

[1] https://en.wikipedia.org/wiki/Decision_tree_learning

[2] http://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting

[3] Popular Ensemble Methods: An Empirical Study, David Opitz, Richard Maclin, Journal of Artificial Intelligence Research (1999)

[4] https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[5] https://www.statsoft.com/Textbook/Boosting-Trees-Regression-Classification/button/1

7 How to use my Project

My project involves working with many datasets. The datasets that I have used are available on the following links:

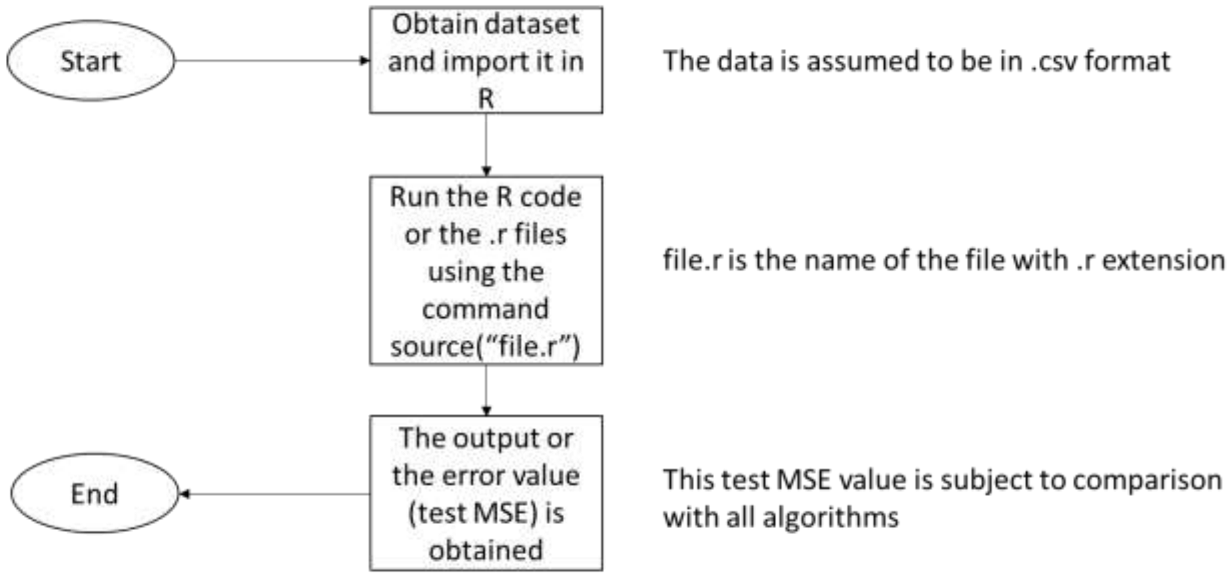
Boston dataset: built-in dataset – can be imported by the command library(MASS)

SkillCraft1 dataset: available on the link –
http://archive.ics.uci.edu/ml/datasets/SkillCraft1+Master+Table+Dataset

The dataset can be downloaded from the Data Folder available on the link. It is a .csv file and can be imported easily into R.

Individual Household electric power consumption dataset: available on the link -
http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption

The dataset can be downloaded from the Data Folder available on the link. It gives a .txt file, which was converted into .csv file for importing it into R.


Start
Obtain dataset and import it in R
The data is assumed to be in .csv format
Run the R code or the .r files using the command source("file.r")
file.r is the name of the file with .r extension
The output or the error value (test MSE) is obtained
This test MSE value is subject to comparison with all algorithms
End

8 Appendix

The code is the same for all datasets, the code used for SkillCraft1 dataset has been included in the appendix. The number of trees generated is kept as B = 1000 for all implementations and in Boosting, the value of the constant η = 0.01.

8.1 Implementation without using Ensemble Methods

```
# To generate a decision tree

library(tree)

Data <- read.table("SkillCraft1_Dataset.csv", header=T,sep=",", na.strings="?")

# The '?' are omitted

Data<-na.omit(Data)


# Training data

train.data <- sample(1:nrow(Data), (nrow(Data)/2))

# Test Data

test <- Data[-train.data,"TotalHours"]


# Generate a tree using tree()

tree.data <- tree(TotalHours~., Data, subset=train.data)

# Predict the output using predict()

yhat <- predict(tree.data, newdata = Data[-train.data,])


# Test MSE is calculated
```

```
test.mse <- (1 / length(test)) * sum((yhat-test)^2)
cat("The test MSE is",test.mse,"\n")


#For Statistical Comparison
for(i in 1:length(test)){
diff[i]<-abs(test[i]-yhat[i])
 }
m<- sample(mean(diff):max(diff),1)
 correct <- 0
 wrong <- 0
 for(i in 1:length(test)){
 if(diff[i]<m)
 correct<-correct+1
 else
 wrong<-wrong+1
 }
cat("The number of times algorithm gives correct values", correct,"\n")
cat("The number of times algorithm gives wrong values", wrong,"\n")
```

8.2 Implementation of Bagging

```
 # To generate a decision tree
library(tree)
Data <- read.table("SkillCraft1_Dataset.csv", header=T,sep=",", na.strings="?")
# The '?' are omitted
Data<-na.omit(Data)


# Training data
```

```
train<-Data[1:(nrow(Data)/2),1:ncol(Data)]

# Test Data

test<-Data[(nrow(Data)/2):nrow(Data),1:ncol(Data)]



B <- 1000

fhat<-matrix(nrow=nrow(test),ncol = B)

fbag<- vector(length = nrow(Data)/2)

y <- test$TotalHours


for(b in 1:B){

 # Training data

 train <- sample(1:nrow(Data), (nrow(Data)/2))

 tree.data <- tree(TotalHours~., Data, subset=train)

 fhat[,b] <- predict(tree.data, newdata = test)

}


for(i in 1:(nrow(Data)/2)){

 fbag[i] <- mean(fhat[i,])

}


sum <- 0

for(i in 1:length(fbag))

{

 sum=sum+(fbag[i]-y[i])^2

}

bagged.mse<-sum/nrow(test)
```

```
cat("The test MSE obtained in bagging is ",bagged.mse,"\n")


#For Statistical Comparison
bagged.correct <- 0
bagged.wrong <- 0


diff<-abs(y-fbag)
m<-sample(mean(diff):max(diff),1)
for(i in 1:nrow(test)){
 if(diff[i]<m)
 bagged.correct<-bagged.correct+1
 else
 bagged.wrong<-bagged.wrong+1
 }


cat("The number of times algorithm gives correct values", bagged.correct,"\n")
cat("The number of times algorithm gives wrong values", bagged.wrong,"\n")
```

8.3 Implementation of Random Forest

```
#Prelimnary steps
Data <- read.table("SkillCraft1_Dataset.csv", header=T,sep=",", na.strings="?")
# The '?' are omitted
Data<-na.omit(Data)


B <- 1000
for(b in 1:B){
training <- sample(nrow(Data), size= (nrow(Data)/2))
```

```
train<-Data[training,]

# Test Data

test<-Data[-training,]

# The actual output

y<-train$TotalHours

y.test <- test$TotalHours

n<-mean(y)

cond <- ifelse(y < n,TRUE,FALSE)

inputs <- train

inputs$TotalHours <- NULL

data <- inputs

data <- na.omit(data)

p <- ncol(data)

m <- floor(sqrt(p))

##m <- p

# dataset_list contains the list of datasets

dataset_list <- list()

checking.condition <- list()

attribute <- 1

x <- root.selection(inputs,p,m,y)

while(attribute != 0){

root <- data[,x[1]]

root <- na.omit(root)

threshold <- x[2]

if(length(dataset_list)!=0)

data <- dataset_list[[1]]

b1 <- vector()
```

```
b2 <- vector()
  j1<-1
  j2<-1
 for(i in 1:length(root)){
  if(root[i]<threshold){
  b1[j1] <- i
  j1<-j1+1
  }
  else{
  b2[j2] <- i
  j2<-j2+1
  }
  }
if(length(dataset_list)!=0)
dataset_list[[1]] <- NULL
dataset_list <- updating.data(data,b1,b2)
checking.condition <- condition(cond,b1,b2)
checking.condition[[1]]<- na.omit(checking.condition[[1]])
checking.condition[[2]]<- na.omit(checking.condition[[2]])
# Checks for the stopping condition
if(check(checking.condition[[1]])==TRUE && check(checking.condition[[2]])==TRUE){
attribute <- 0
} else{
attribute <- 1
}
for(i in 1: length(dataset_list)){
cond <- checking.condition[[i]]
cond <- na.omit(cond)

correctness <- check(cond)

# Checks for the purity of the condition

if (correctness == FALSE){

data <- dataset_list[[i]]

data <- na.omit(data)

if(nrow(data)!=0)

x <- root.selection(dataset_list[[i]],p,m,y)

dataset_list[length(dataset_list)+1] <- data

}

data <- na.omit(data)

if(nrow(data)==0)

attribute <- 0

}

}

yless[b] <- x[3]

ymore[b] <- x[4]

}

yless <- na.omit(yless)

yless <- mean(yless)

ymore <- na.omit(ymore)

ymore <- mean(ymore)

n <- mean(y.test)

rss <- 0

for(i in 1:nrow(test)){

if(y.test[i] < n){

rss[i] <- ((y.test - yless)^2)
```

```
} else{
rss[i] <- ((y.test - ymore)^2)
}
}
test.mse<- mean(rss)
cat("The test MSE obtained in random forest is",test.mse,"\n")

# For Statiscal Comparison
yhat <- (yless+ymore)/2
diff<-abs(y-yhat)
 forest.correct <- 0
 forest.wrong <- 0
 m<-sample(mean(diff):max(diff),1)
 for(i in 1:nrow(test)){
  if(diff[i]<m)
 forest.correct <- forest.correct + 1
 else
 forest.wrong <- forest.wrong + 1
}
cat ("The number of times random forest gives correct values",forest.correct,"\n")
cat("The number of times random forest gives wrong values",forest.wrong,"\n")
```

8.3.1 Functions used in the Implementation of Random Forest

8.3.1.1 Root Selection Function

```
root.selection <- function(data, p, m,y){
attributes <- matrix(,nrow(data),m)
#attribute numbers are chosen
a<- sample(p,m)
```

```
for(i in 1:m){
attributes[,i] <- data[,a[i]]
}
attributes <- na.omit(attributes)
# the range of the attributes
# each column of the range matrix indicate an attribute among the ‘m’ attributes
# The first row of range matrix contains the maximum value of m attributes
# The second row of range matrix contains the minimum value of m attributes
range <- matrix(nrow=m,ncol=2)
#diff <- vector(length=m)
for(i in 1:m){
range[i,1] <-  floor(max(attributes[,i]))
range[i,2] <-  ceiling(min(attributes[,i]))
}
range <- na.omit(range)
best.attribute <- 0
threshold <- 0
left.yhat <- 0
right.yhat <- 0
minimum.rss <- Inf
leaf.indicator <- 0
  for (j in 1:m) {
  # diff <- abs(range[j,1] - range[j,2])/nrow(train)
   s <- seq(range[j,2],range[j,1])
   for(t in 1 : length(s)) {
   # yless is the left side prediction
   # ymore is the right side prediction
# l is the counter for values less than the threshold
# m is the counter for values greater than the threshold
  yless <- 0
  l <- 0
  ymore <- 0
  m <- 0
  for (i in 1:nrow(attributes)) {
    if (attributes[i,j]<s[t]) {
      yless <- yless + y[i]
      l <- l + 1
    }
    else {
      ymore <- ymore + y[i]
      m <- m + 1
    }
  }
  yless <- yless / l
  ymore <- ymore / m

  rss <- 0
  for (i in 1:nrow(attributes)) {
    if (attributes[i,j]<s[t]) {
      rss <- rss + (yless - y.train[i])^2
    }
    else {
      rss <- rss + (ymore - y.train[i])^2
    }
```

```
      }
if(rss < minimum.rss){
left.yhat <- yless
right.yhat <- ymore
threshold <- s[t]
best.attribute <- a[j]
minimum.rss <- rss
}
} # ends loop for s
} # ends loop for m or attributes
return(c(best.attribute, threshold, left.yhat, right.yhat))
}
```

8.3.1.2 Updating dataset Function

```
updating.data <- function(data,b1,b2){
for(i in 1:length(b1)){
d1[i,]<-data[b1[i],]
}
for(i in 1:length(b2)){
 d2[i,]<-data[b2[i],]
}
l <- list(d1,d2)
return(l)
}
```

8.3.1.3 Defining Stopping Condition

```
condition <- function(cond,b1,b2){
for(i in 1:length(b1)){
```

```
c1[i] <- cond[b1[i]]
}
for(i in 1:length(b2)){
c2[i] <- cond[b2[i]]
}
c <- list(c1,c2)
return (c)
}
```

8.3.1.4 Checking the purity of the condition

```
check <- function(cond){
if(all(cond==cond[1])){
return (TRUE)
} else{
return (FALSE)
}
}
```

8.4 Implementation of Boosting

```
#Prelimnary steps
Data <- read.table("SkillCraft1_Dataset.csv", header=T,sep=",", na.strings="?")
# The '?' are omitted
Data<-na.omit(Data)


# learning rate
eta <- 0.01
# the number of decision trees
```

```
B <- 1000


# Training data

train<-Data[1:(nrow(Data)/2),1:ncol(Data)]

# Test Data

test<-Data[(nrow(Data)/2):nrow(Data),1:ncol(Data)]

# The actual output

y.train<-train$TotalHours

# The test label

y.test<-test$TotalHours


# the attributes in training set

x.train <- matrix(nrow=nrow(train),ncol=4)

j <- 1

for(i in 6:7){

x.train[,j] <- train[,i]

j<-j+1

}

# the attributes in test set

x.test <- matrix(nrow= nrow(test),ncol=4)

j <- 1

for(i in 6:7){

x.test[,j] <- test[,i]

j<-j+1

}


# the range of the attributes
```

```
range <- matrix(nrow=4,ncol=2)

for(i in 1:2){

range[i,1] <- floor(max(x.train[,i]))

range[i,2] <- ceiling(min(x.train[,i]))

}

# the first column of decision_stump matrix contains the input attribute number

# the second column of decision_stump matrix contains the threshold value which

# produces the minimum training rss

#the third column of decision_stump matrix contains best yless

#the fourth column of decision_stump matrix contains best ymore

decision_stump <- matrix(nrow=B,ncol=4)

# Initialising residuals

r <- y.train

for (b in 1:B) {

 attribute <- 0

 threshold <- 0

 left.yhat <- 0

 right.yhat <- 0

 minimum.rss <- Inf

 for (j in 1:2) {

  ##diff <- abs(range[j,1] - range[j,2])/nrow(train)

  s <- seq(range[j,2],range[j,1])

  for(t in 1 : length(s)) {

  # yless is the left side prediction

  # ymore is the right side prediction

  # l is the counter for values less than the threshold

  # m is the counter for values greater than the threshold
```

```
      yless <- 0
      l <- 0
      ymore <- 0
      m <- 0
      for (i in 1:nrow(train)) {
        if (x.train[i,j]<s[t]) {
          yless <- yless + r[i]
          l <- l + 1
        }
        else {
          ymore <- ymore + r[i]
          m <- m + 1
        }
      }
      yless <- yless / l
      ymore <- ymore / m

      rss <- 0
      for (i in 1:nrow(train)) {
        if (x.train[i,j]<s[t]) {
          rss <- rss + (yless - r[i])^2
        }
        else {
          rss <- rss + (ymore - r[i])^2
        }
      }
if(rss < minimum.rss){
```

```
left.yhat <- yless
right.yhat <- ymore
threshold <- s[t]
attribute <- j
minimum.rss <- rss
}
}
}
 decision_stump[b,1] <- attribute
 decision_stump[b,2] <- threshold
 decision_stump[b,3] <- left.yhat
 decision_stump[b,4] <- right.yhat
 # update residuals
 for (i in 1:nrow(train)) {
  if (x.train[i,attribute]<threshold) {
   r[i] <- r[i] - eta * left.yhat
  }
  else {
   r[i] <- r[i] - eta * right.yhat
  }
 }
}
train.rss <- vector(length=nrow(train))
test.rss <- vector(length=nrow(train))
for (i in 1:nrow(train)) {
 yhat <- 0
 for (b in 1:B) {
    if (x.train[i,decision_stump[b,1]]<decision_stump[b,2]) {
      yhat <- yhat + eta*decision_stump[b,3]
    }
    else {
      yhat <- yhat + eta*decision_stump[b,4]
    }
  }
  train.rss[i] <- (yhat - y.train[i])^2
}
train.MSE <- mean(train.rss)


for (i in 1:nrow(test)) {
  yhat <- 0
  for (b in 1:B) {
    if (x.test[i,decision_stump[b,1]] < decision_stump[b,2]) {
      yhat <- yhat + eta * decision_stump[b,3]
    }
    else {
      yhat <- yhat + eta*decision_stump[b,4]
    }
  }
  test.rss[i] <- (yhat - y.test[i])^2
}
test.MSE <- mean(test.rss)
cat("The training MSE is",train.MSE,"\n")
cat("The test MSE is",test.MSE,"\n")
```